100 XP

Introduction

10 minutes

In Microsoft Power BI, the data model underpins report and dashboard development. To produce the right reports and dashboards, you should have a clear understanding of the questions that your report and dashboard users will ask. Your next focus should be on the data model design that will best support the visualizations in your reports and dashboards. After you've arrived at a model design, you're ready to use Power BI Desktop to develop the model.

When developing the model, you will complete the following tasks:

- Connect to data
- Transform and prepare data
- Define business logic by adding Data Analysis Expressions (DAX) calculations
- Enforce data permissions with row-level security by adding roles
- Publish the model to Power BI

Understanding the structure of data models can help you design the right model to support your reports and dashboards. A data model can be developed in many ways, yet one or several of those ways are more optimal. Optimal models are important for delivering good query performance and for minimizing data refresh times and the use of service resources, including memory and CPU. The fewer resources that are used, the more models that can be hosted and at lower cost.

Watch the following video to learn about what makes up a data model that is developed in Power BI Desktop.
https://www.microsoft.com/en-us/videoplayer/embed/RE4AxcV?postJsllMsg=true

Next unit: Star schema design

Continue

Star schema design

5 minutes

It's unusual for a Power BI data model to be comprised of a single table. A single-table model can be a simple design, perhaps one that's suitable for a data exploration task or proof of concept, but not one that's an optimal model design. An optimal model adheres to *star schema* design principles. Star schema refers to a design approach that's commonly used by relational data warehouse designers because it presents a user-friendly structure and it supports high-performance analytic queries.

This design principle is called a star schema because it classifies model tables as either *fact* or *dimension*. In a diagram, a fact table forms the center of a star, while dimension tables, when placed around a fact table, represent the points of the star.

Fact tables

The role of a fact table is to store an accumulation of rows that represent observations or events that record a specific business activity. For example, events that are stored in a sales fact table could be sales orders and the order lines. You could also use a fact table to record stock movements, stock balances, or daily currency exchange rates. Generally, fact tables contain numerous rows. As time passes, fact table rows accumulate. In analytic queries (which will be defined later in this module), fact table data is summarized to produce values like sales and quantity.

Dimension tables

Dimension tables describe your business entities, which commonly represent people, places, products, or concepts. A date dimension table, which contains one row for each date, is a common example of a concept dimension table. The columns in dimension tables allow filtering and grouping of fact table data.

Each dimension table must have a unique column, which is referred to as its key column. A unique column doesn't contain duplicate values and it should never have missing values. In a product dimension table, the column could be named **ProductKey** or **ProductID**. Likely, additional columns will store descriptive values, like the product name, subcategory, category, color, and so on. In analytic queries, these columns are used to filter and group data.

Compare fact and dimension tables

The following figure compares characteristics of fact and dimension tables.

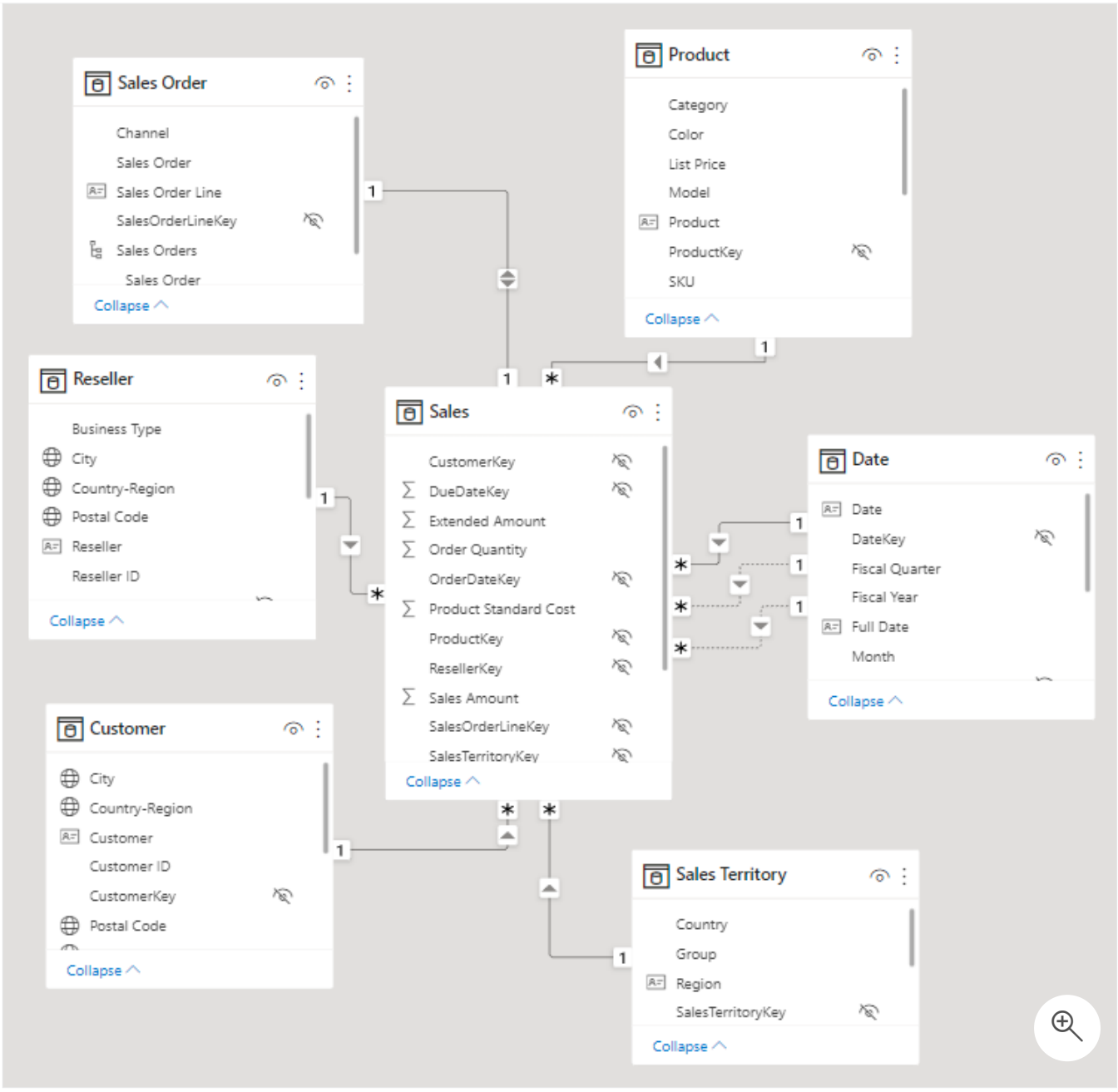
Characteristic	Dimension table	Fact table
Model purpose	Stores business entities	Stores events or observations
Table structure	Includes a key column and descriptive columns for filtering and grouping	Includes dimension key columns and numeric measure columns that can be summarized
Data volume	Typically, contains fewer rows (relative to fact tables)	Can contain numerous rows
Query purpose	To filter and group	To summarize

Relate star schema tables

In the model, dimension tables are related to fact tables by using one-to-many relationships. The relationships allow filters and groups that are applied to dimension table columns to propagate to the fact table. This design pattern is common.

Dimension tables can be used to filter multiple fact tables, and fact tables can be filtered by multiple dimension tables. However, it's not a good practice to relate a fact table directly to another fact table.

To practice this concept, download the **Adventure Works DW 2020 M01.pbix** file, open the file, and then switch to the model diagram.

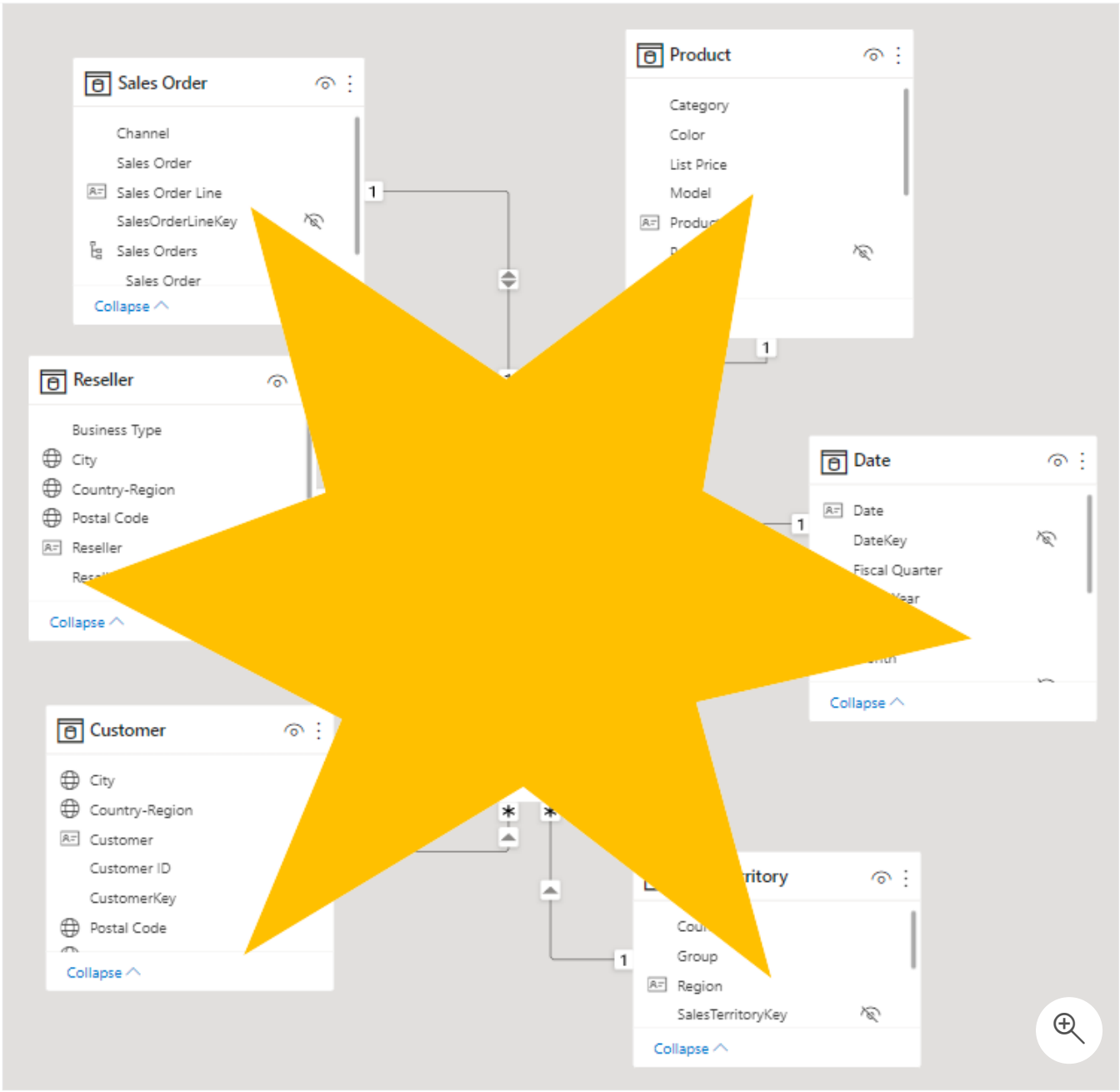


Notice that the model is comprised of seven tables, one of which is named **Sales** and is the fact table. The remaining tables are dimension tables, and they have the following names:

- Customer
- Date
- Product
- Reseller
- Sales Order
- Sales Territory

Notice the relationships between the dimension and fact tables and that each relationship filter direction is pointing toward the fact table. As a result, when filters are applied to dimension table columns (to filter or group by column values), related facts are filtered and summarized.

If you examine the pattern, you might see a star shape.



For more information on star schema design, see Understand star schema and the importance for Power BI.

Next unit: Analytic queries

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

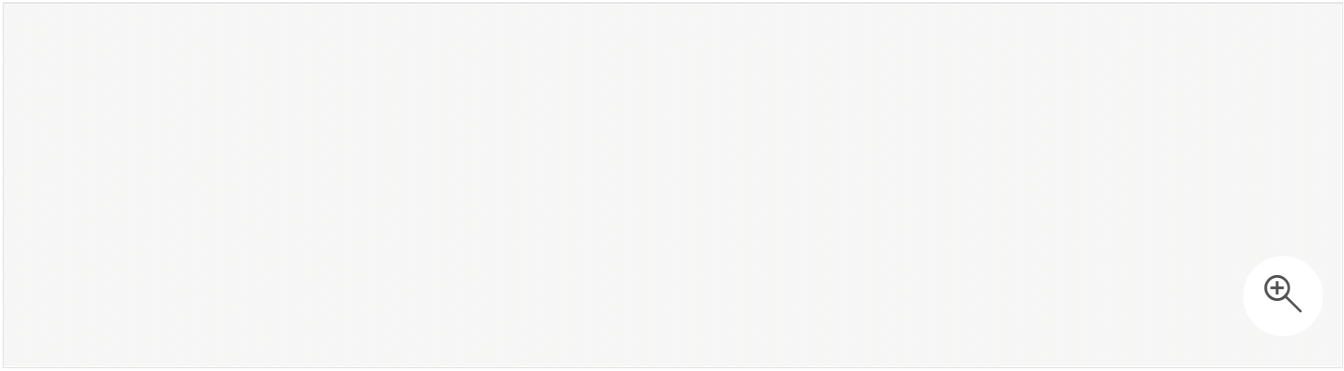
Analytic queries

2 minutes

An *analytic query* is a query that produces a result from a data model. Each Power BI visual, in the background, submits an analytic query to Power BI to query the model. The analytic query is written as a Data Analysis Expressions (DAX) query statement. However, you don't need to write a native DAX statement; you only need to configure report visuals by mapping dataset fields.

An analytic query has three phases that are implemented in the following order:

1. Filter
2. Group
3. Summarize



Filtering, or slicing, targets the data of relevance. In Power BI reports, filters can be applied to three different scopes: the entire report, a specific page, or a specific visual. Filtering is also applied in the background when row-level security (RLS) is enforced. Each report visual can inherit filters or have filters directly applied to it.

Grouping, or dicing, divides query results into groups.

Summarizing produces a single value result. Typically, numeric columns are summarized by using summarization methods (sum, count, and many others). These methods are simple summarizations. More complex summarizations, like a percent of grand total, can be achieved by defining measures that are written in DAX.

Not all analytic queries need to filter, group, and summarize:

- Commonly, report visuals are filtered, perhaps by a time period or geographic location.
- Grouping is optional. For example, a card visual, which is used to display a single value, isn't concerned with grouping.

- Typically, report visuals summarize. One notable exception, however, is the slicer visual, which isn't concerned with summarization.

Next unit: Configure report visuals

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Configure report visuals

5 minutes

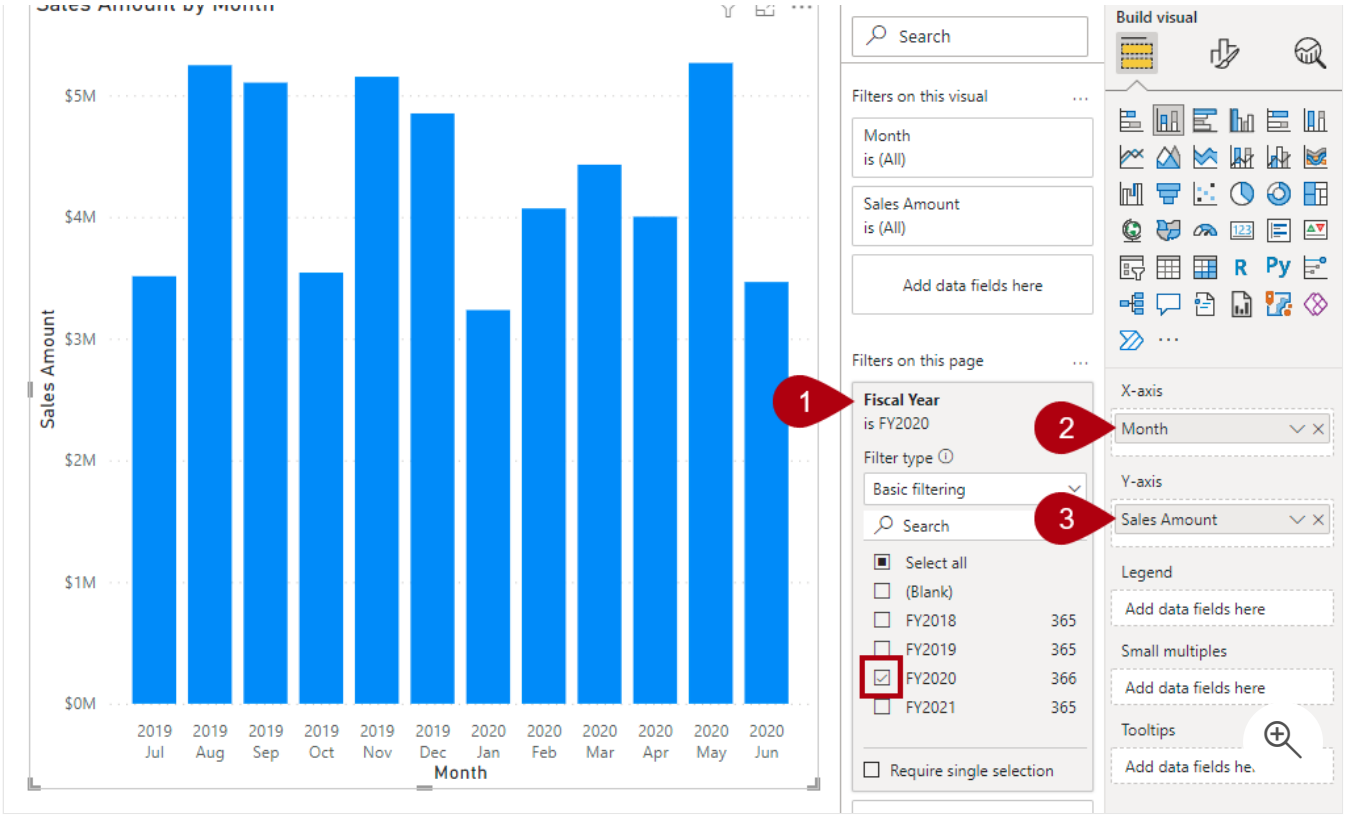
Report authors produce report designs by adding report visuals and other elements to pages. Other elements include text boxes, buttons, shapes, and images. Each of these elements is configured independently of dataset fields.

At design time, adding and configuring a report visual involves the following methodology:

1. Select a visual type, like a bar chart.
2. Map dataset fields, which are displayed in the **Fields** pane, to the visual field wells. For a bar chart, the wells are **Y-axis**, **X-axis**, **Legend**, **Small multiples**, and **Tooltips**.
3. Configure mapped fields. It's possible to rename mapped fields or toggle the field to summarize or not summarize. If the field summarizes, you can select the summarization method.
4. Apply format options, like axis properties, data labels, and many others.

The following example shows how to configure the analytic query for a report visual. To begin, open the Adventure Works DW 2020 M01.pbix Power BI Desktop file, and then follow these steps:

1. Add a stacked column chart visual to the report page.
2. **Filter** the page by using **Fiscal Year** from the **Date** table and selecting **FY2020**.
3. **Group** the visual by adding **Month** from the **Date** table to the **X-axis** well.
4. **Summarize** the visual by adding **Sales Amount** from the **Sales** table to the **Y-axis** well.



Fields is a collective term that is used to describe a model resource *that can be used to configure a visual*. The three different model resources that are fields include:

- Columns
- Hierarchy levels
- Measures

Each of these resource types can be used to configure a visual, which in the background configures an analytic query. The following table illustrates how to use each of the model resources.

Model resource	Filter	Group	Summarize
Column	X	X	X
Hierarchy level	X	X	
Measure	X		X

Columns

Use columns to filter, group, and summarize column values. Summarizing numeric columns is

common, and it can be done by using sum, count, distinct count, minimum, maximum, average, median, standard deviation, or variance. You can also summarize text columns by using first (alphabetic order), last, count, or distinct count. Additionally, you can summarize date columns by using earliest, latest, count, or distinct count.

At design time, the data modeler can set the column default summarization property. This property can be set to any of the supported summarization types or to **Do not summarize**. The latter option means that the column, by default, is only to be used to group. If your data model has a numeric column that stores year values, it would be appropriate to set its default summarization to **Do not summarize** because the column will likely be used only for grouping or filtering, and that numeric summarization of years, like an average, doesn't produce a meaningful result.

Hierarchy levels

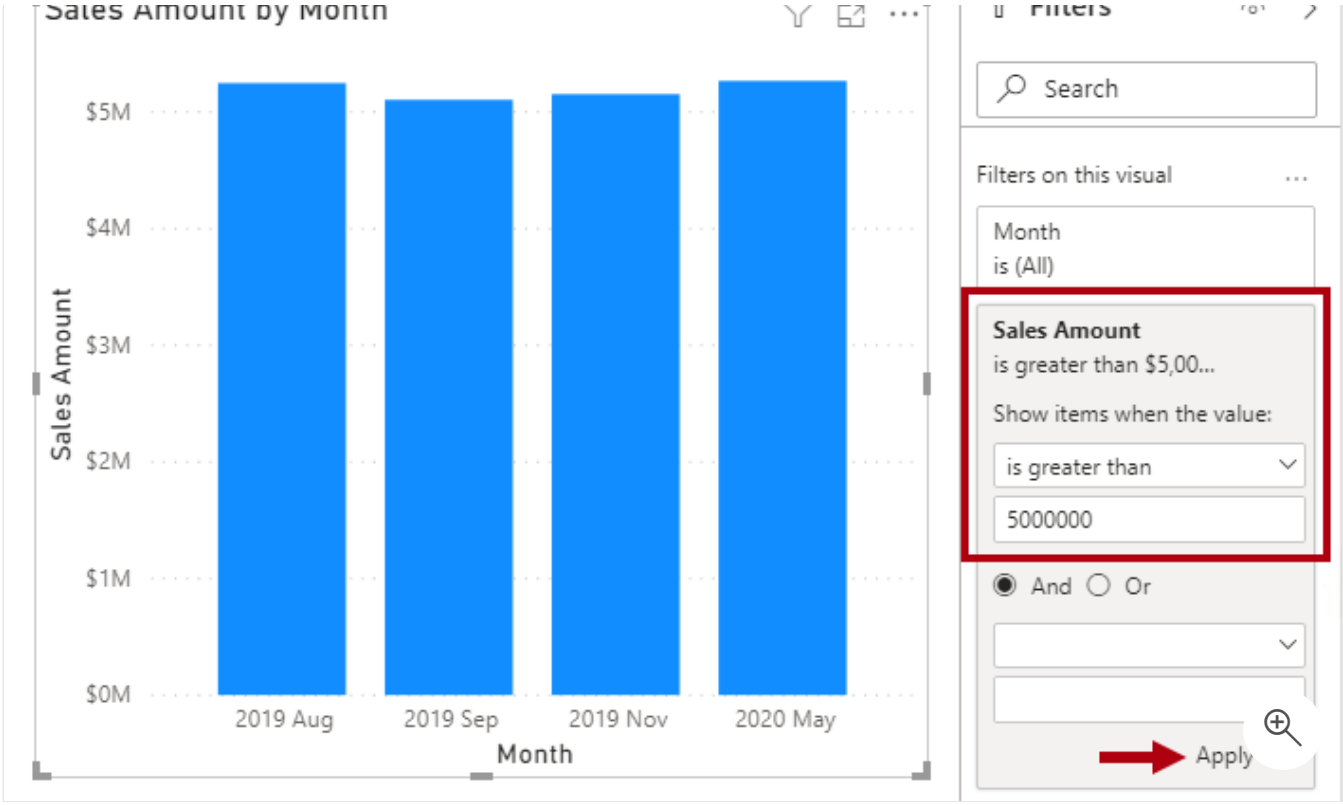
While hierarchy levels are based on columns, they can be used to filter and group but not to summarize. Report authors can summarize the column that the hierarchy level is based on, provided that it's visible in the **Fields** pane.

Measures

Measures are designed to summarize model data; they can't be used to group data. However, measures can be used to filter data in one special case: to use a measure to filter a visual when the visual displays the measure and the filter is a visual-level filter (so, not a report or page-level filter). When used in this way, a measure filter is applied *after* the analytic query has summarized data. This process is done to eliminate groups where the measure filter condition isn't true. (For those who are familiar with SQL syntax, a measure that used to filter a visual is like the `HAVING` clause in a `SELECT` statement.)

The following figure shows the stacked column visual adjusted to display groups (months) when sales amounts exceed $5 million. This adjustment is done in the **Filters** pane by applying a filter to the **Sales Amount** field: Configure the filter to show items when the value *is greater than 5,000,000*. Remember to select **Apply filter**, which is located in the lower-right corner of the card.

Notice that only four groups (months) have sales amounts exceeding $5 million.



Next unit: Check your knowledge

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

200 XP

Check your knowledge

6 minutes

Answer the following questions to see what you've learned.

1. In a Power BI Desktop model design, which type of object do you create to enforce row-level security? *

- ( ) Table
  - ✗ A table, which is defined by columns, stores rows of data. It does not enforce row-level security.
- ( ) Column
- ( ) Measure
- ( ) Role
  - ✓ A role consists of one or more rules, which are DAX formulas that are used to filter table rows. Roles define row-level security.

2. Which of the following statements is correct regarding a star schema design? *

- ( ) Fact tables store accumulations of business events.
  - ✓ Fact tables store accumulations of business events, like sales orders or currency exchange rates.
- ( ) Fact tables store accumulations of business entities.
  - ✗ Fact tables store accumulations of business events. They do not store business entities, which are stored in dimension tables.
- ( ) Fact tables must have a unique column.

3. In what order does an analytic query implement its phases? *

Filter, Group, Summarize

✔ Filter is used to first restrict the data to query. Group then divides the query result into groups. Summarize then produces single value aggregations for each group.

Group, Filter, Summarize

✘ Filter takes place before grouping.

Summarize, Filter, Group

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Introduction

5 minutes

By using Data Analysis Expressions (DAX), you can add three types of calculations to your data model:

- Calculated tables
- Calculated columns
- Measures

> **Note**
>
> DAX can also be used to define row-level security (RLS) rules, which are expressions that enforce filters over model tables. However, rules aren't considered to be model calculations so they're out of scope for this module. For more information, see **Row-level security (RLS) with Power BI**.

Calculated tables

You can write a DAX formula to add a calculated table to your model. The formula can duplicate or transform *existing model data*, or create a series of data, to produce a new table. Calculated table data is always imported into your model, so it increases the model storage size and can prolong data refresh time.

> **Note**
>
> A calculated table can't connect to external data; you need to use Power Query to accomplish that task.

Calculated tables can be useful in various scenarios:

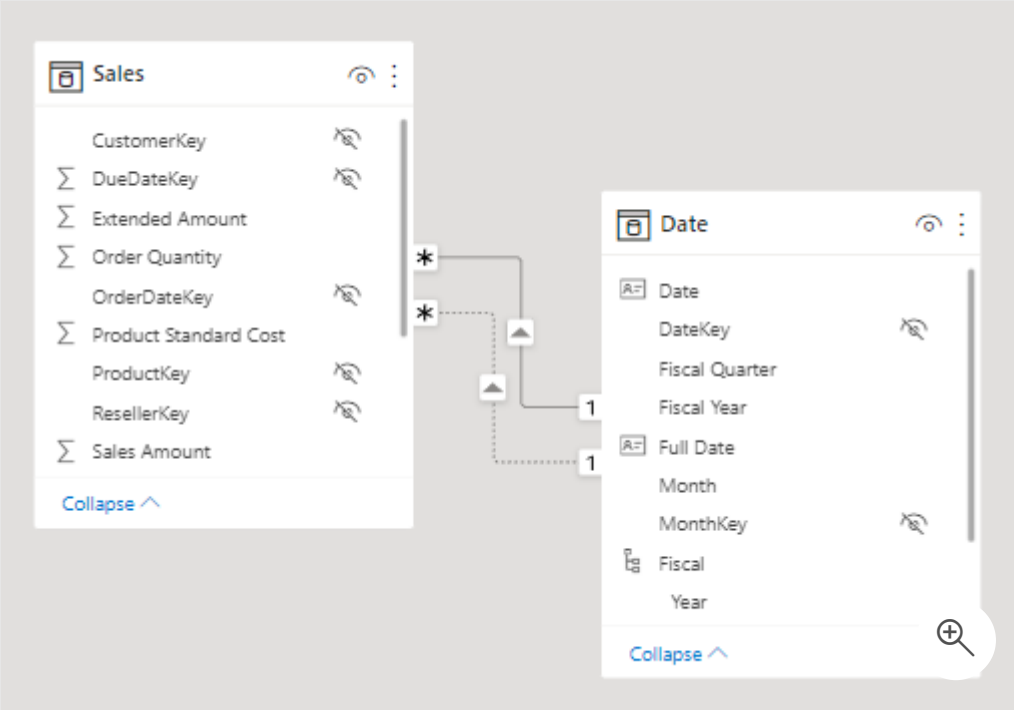
- Date tables
- Role-playing dimensions
- What-if analysis

Date tables

Date tables are required to apply special time filters known as *time intelligence*. DAX time intelligence functions only work correctly when a date table is set up. When your source data doesn't include a date table, you can create one as calculated tables by using the CALENDAR or CALENDARAUTO DAX functions.

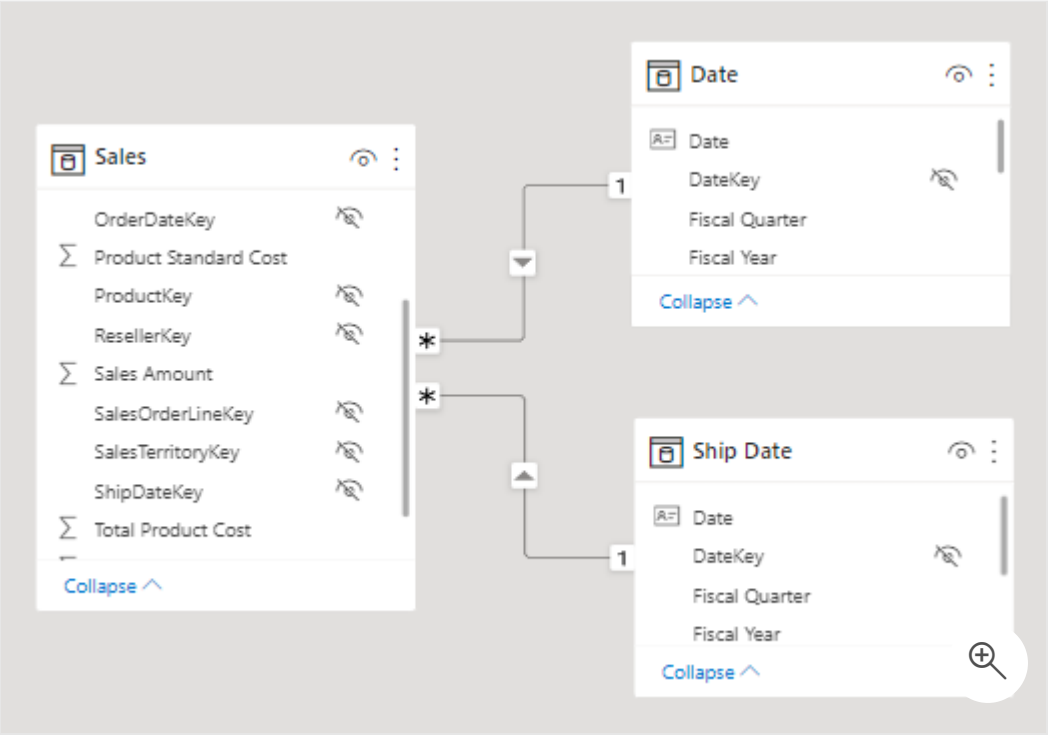
Role-playing dimensions

When two model tables have multiple relationships, it could be because your model has a role-playing dimension. For example, if you have a table named **Sales** that includes two date columns, **OrderDateKey** and **ShipDateKey**, both columns are related to the **Date** column in the **Date** table. In this case, the **Date** table is described as a role-playing dimension because it could play the role of *order date* or *ship date*.



Microsoft Power BI models only allow one active relationship between tables, which in the model diagram is indicated as a solid line. The active relationship is used by default to propagate filters, which in this case would be from the **Date** table to the **OrderDateKey** column in the **Sales** table. Any remaining relationships between the two tables are inactive. In a model diagram, the relationships are represented as dashed lines. Inactive relationships are only used when they're expressly requested in a calculated formula by using the USERELATIONSHIP DAX function.

Perhaps a better model design could have two date tables, each with an active relationship to the **Sales** table. Thus, report users can filter by order date or ship date, or both at the same time. A calculated table can duplicate the **Date** table data to create the **Ship Date** table.



What-if analysis

Power BI Desktop supports a feature called What-if parameters. When you create a what-if parameter, a calculated table is automatically added to your model.

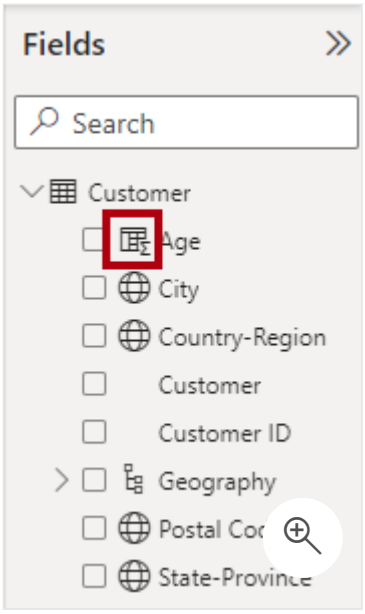
What-if parameters allow report users to select or filter by values that are stored in the calculated table. Measure formulas can use selected value(s) in a meaningful way. For example, a **what-if** parameter could allow the report user to select a hypothetical currency exchange rate, and a measure could divide revenue values (in a local currency) by the selected rate.

Notably, what-if calculated tables aren't related to other model tables because they're not used to propagate filters. For this reason, they're sometimes called *disconnected tables*.

Calculated columns

You can write a DAX formula to add a calculated column to any table in your model. The formula is evaluated for each table row and it returns a single value. When added to an Import storage mode table, the formula is evaluated when the data model is refreshed, and it increases the storage size of your model. When added to a DirectQuery storage mode table, the formula is evaluated by the underlying source database when the table is queried.

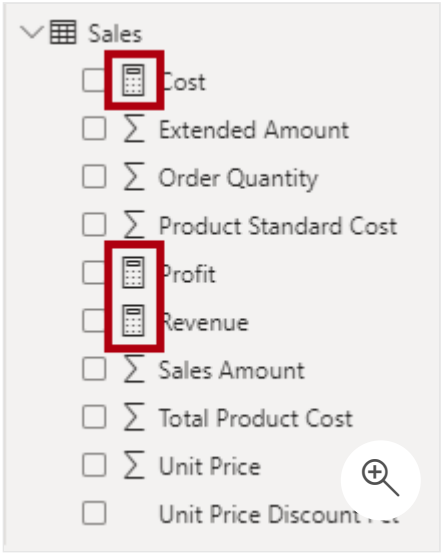
In the **Fields** pane, calculated columns are enhanced with a special icon. The following example shows a single calculated column in the **Customer** table called **Age**.



Measures

You can write a DAX formula to add a measure to any table in your model. The formula is concerned with achieving summarization over model data. Similar to a calculated column, the formula must return a single value. Unlike calculated columns, which are evaluated at data refresh time, measures are evaluated at query time. Their results are never stored in the model.

In the **Fields** pane, measures are shown with the calculator icon. The following example shows three measures in the **Sales** table: **Cost**, **Profit**, and **Revenue**.



Occasionally, measures can be described as *explicit measures*. To be clear, explicit measures are model calculations that are written in DAX and are commonly referred to as simply *measures*. Yet, the concept of *implicit measures* exists, too. Implicit measures are columns that can be summarized by visuals in simplistic ways, like count, sum, minimum, maximum, and so on. You can identify implicit measures in the **Fields** pane because they're shown with the sigma symbol (∑).

> **Note**
>
> Any column can be summarized when added to a visual. Therefore, whether they're shown with the sigma symbol or not, when they're added to a visual, they can be set up as implicit measures.

Additionally, no such concept as a *calculated measure* exists in tabular modeling. The word *calculated* is used to describe calculated tables and calculated columns, which distinguishes them from tables and columns that originate from Power Query. Power Query doesn't have the concept of an explicit measure.

Next unit: Write DAX formulas

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Write DAX formulas

6 minutes

Each model calculation type, calculated table, calculated column, or measure is defined by its name, followed by the equals symbol (=), which is then followed by a DAX formula. Use the following template to create a model calculation:

```DAX
<Calculation name> = <DAX formula>
```

For example, the definition of the **Ship Date** calculated table that duplicates the **Date** table data is:

```DAX
Ship Date = 'Date'
```

A DAX formula consists of expressions that return a result. The result is either a table object or a scalar value. Calculated table formulas must return a table object; calculated column and measure formulas must return a scalar value (single value).

Formulas are assembled by using:

- DAX functions
- DAX operators
- References to model objects
- Constant values, like the number 24 or the literal text "FY" (abbreviation for fiscal year)
- DAX variables
- Whitespace

Tip

When entering DAX formulas in Power BI Desktop, you have the benefit of *IntelliSense*. IntelliSense is a code-completion aid that lists functions and model resources. When you select a DAX function, it also provides you with a definition and description. We recommend that you use IntelliSense to help you quickly build accurate formulas.

DAX functions

Similar to Microsoft Excel, DAX is a functional language meaning that formulas rely on functions to accomplish specific goals. Typically, DAX functions have arguments that allow passing in variables. Formulas can use many function calls and will often nest functions within other functions.

In a formula, function names must be followed by parentheses. Within the parentheses, variables are passed in.

> **Note**
>
> Some functions don't take arguments, or arguments might be optional.

Working with DAX functions is described later in this module.

DAX operators

Formulas also rely on operators, which can perform arithmetic calculations, compare values, work with strings, or test conditions.

DAX operators are described in more detail later in this module.

References to model objects

Formulas can only refer to three types of model objects: tables, columns, or measures. A formula can't refer to a hierarchy or a hierarchy level. (Recall that a hierarchy level is based on a column, so your formula can refer to a hierarchy level's column.)

Table references

When you reference a table in a formula, officially, the table name is enclosed within single quotation marks. In the following calculated table definition, notice that the **Date** table is enclosed within single quotation marks.

DAX

```
Ship Date = 'Date'
```

However, single quotation marks can be omitted when both of the following conditions are true:

1. The table name does not include embedded spaces.
2. The table name isn't a reserved word that's used by DAX. All DAX function names and operators are reserved words. *Date* is a DAX function name, which explains why, when you are referencing a table named **Date**, that you must enclose it within single quotation marks.

In the following calculated table definition, it's possible to omit the single quotation marks when referencing the **Airport** table:

```
DAX
Arrival Airport = Airport
```

Column references

When you reference a column in a formula, the column name must be enclosed within square brackets. Optionally, it can be preceded by its table name. For example, the following measure definition refers to the **Sales Amount** column.

```
DAX
Revenue = SUM([Sales Amount])
```

Because column names are unique within a table but not necessarily unique within the model, you can disambiguate the column reference by preceding it with its table name. This disambiguated column is known as a *fully qualified column*. Some DAX functions require passing in fully qualified columns.

> **Tip**
>
> To improve the readability of your formulas, we recommend that you always precede a column reference with its table name.

The previous example measure definition can be rewritten as:

```
DAX
Revenue = SUM(Sales[Sales Amount])
```

Measure references

When you reference a measure in a formula, like column name references, the measure name must be enclosed within square brackets. For example, the following measure definition refers to the **Revenue** and **Cost** measures.

```
DAX
Profit = [Revenue] - [Cost]
```

If you're a DAX beginner, the fact that column and measure references are always enclosed within square brackets can cause confusion when you're trying to read a formula. However, as you become proficient with DAX fundamentals, you'll be able to determine which type of object it is because, in DAX formulas, columns, and measures are used in different ways.

> **Tip**
>
> It's possible to precede a measure reference with its table name. However, measures are a model-level object. While they're assigned to a home table, it's only a cosmetic relationship to logically organize measures in the **Fields** pane.
>
> Therefore, while we recommend that you always precede a column reference with its table name, the inverse is true for measures: We recommend that you never precede a measure reference with its table name.
>
> For more information, see **Column and measure references**.

DAX variables

Formulas can declare DAX variables to store results.

How and when to use DAX variables is described later in this module.

Whitespace

Whitespace refers to characters that you can use to format your formulas in a way that's quick and simple to understand. Whitespace characters include:

- Spaces
- Tabs
- Carriage returns

Whitespace is optional and it doesn't modify your formula logic or negatively impact performance. We strongly recommend that you adopt a format style and apply it consistently, and consider the following recommendations:

- Use spaces between operators.
- Use tabs to indent nested function calls.
- Use carriage returns to separate function arguments, especially when it's too long to fit on a single line. Formatting in this way makes it simpler to troubleshoot, especially when the formula is missing a parenthesis.
- Err on the side of too much whitespace than too little.

> **Tip**
>
> In the formula bar, to enter a carriage return, press **Shift+Enter**. Pressing **Enter** alone will commit your formula.

Consider the following measure definition that's written in a single line and that includes five DAX function calls:

DAX

```
Revenue YoY % = DIVIDE([Revenue] - CALCULATE([Revenue],
SAMEPERIODLASTYEAR('Date'[Date])), CALCULATE([Revenue],
SAMEPERIODLASTYEAR('Date'[Date])))
```

The following example is the same measure definition but now formatted, which helps make it easier to read and understand:

DAX

```
Revenue YoY % =
DIVIDE(
    [Revenue]
        - CALCULATE(
            [Revenue],
            SAMEPERIODLASTYEAR('Date'[Date])
    ),
    CALCULATE(
        [Revenue],
        SAMEPERIODLASTYEAR('Date'[Date])
    )
)
```

Try formatting the measure on your own. Open the Adventure Works DW 2020 M02.pbix Power BI Desktop file and then, in the **Fields** pane, expand the **Sales** table and then select the

Revenue YoY % measure. In the formula bar, use tab and carriage return characters to produce the same result as the previous example. When you add a carriage return, remember to press **Shift+Enter**.

This measure definition can be further improved for readability and performance, which will be explained later in this module.

> **Tip**
>
> An excellent formatting tool from another source that can help you format your calculations is **DAX Formatter** . This tool allows you to paste in your calculation and format it. You can then copy the formatted calculation to the clipboard and paste it back into Power BI Desktop.

Next unit: DAX data types

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

DAX data types

2 minutes

Data model columns have a set data type, which ensures that all column values conform to that data type. Column data types are defined in Power Query, or in the case of calculated columns, it's inferred from the formula. Measure data types, similar to calculated column data types, are inferred from the formula.

Model data types aren't the same as DAX data types, though a direct relationship exists between them. The following table lists the model data types and DAX data types. Notice the supported range of values for each data type.

Model data type	DAX data type	Description
Whole number	64-bit integer	-2^{63} through $2^{63}-1$
Decimal number	64-bit real	Negative: -1.79×10^{308} through -2.23×10^{-308} - zero (0) - positive: 2.23×10^{-308} through 1.79×10^{308} - Limited to 17 decimal digits
Boolean	Boolean	TRUE or FALSE
Text	String	Unicode character string
Date	Date/time	Valid dates are all dates after March 1, 1900
Currency	Currency	-9.22×10^{14} through 9.22×10^{14} - limited to four decimal digits of fixed precision
N/A	BLANK	In some cases, it's the equivalent of a database (SQL) NULL

BLANK data type

The BLANK data type deserves a special mention. DAX uses BLANK for both database NULL and for blank cells in Excel. BLANK doesn't mean zero. Perhaps it might be simpler to think of it as the *absence of a value*.

Two DAX functions are related to the BLANK data type: the BLANK DAX function returns BLANK, while the ISBLANK DAX function tests whether an expression evaluates to BLANK.

Next unit: Work with DAX functions

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Work with DAX functions

3 minutes

The DAX function library consists of hundreds of functions, each designed to accomplish a specific goal.

Because DAX originated with the Power Pivot add-in for Microsoft Excel 2010, over 80 functions are available that can also be found in Excel. It was a deliberate design strategy by Microsoft to ensure that Excel users can quickly become productive with DAX.

However, many functions exist that you won't find in Excel because they're specific to data modeling:

- Relationship navigation functions
- Filter context modification functions
- Iterator functions
- Time intelligence functions
- Path functions

> **Tip**
>
> To search for documentation that is related to a DAX function, in a web search, enter the keyword *DAX* followed by the function name.

For more information, see the DAX function reference.

Functions that originate from Excel

The following sections consider several useful functions that you might already be familiar with because they exist in Excel.

The IF DAX function tests whether a condition that's provided as the first argument is met. It returns one value if the condition is TRUE and returns the other value if the condition is FALSE. The function's syntax is:

DAX

```
IF(<logical_test>, <value_if_true>[, <value_if_false>])
```

> **Tip**
>
> A function argument is optional when documentation shows it enclosed within square brackets.

If **logical_test** evaluates to FALSE and **value_if_false** isn't provided, the function will return BLANK.

Many Excel summarization functions are available, including `SUM`, `COUNT`, `AVERAGE`, `MIN`, `MAX`, and many others. The only difference is that in DAX, you pass in a column reference, whereas in Excel, you pass in a range of cells.

Many Excel mathematic, text, date and time, information, and logical functions are available as well. For example, a small sample of Excel functions that are available in DAX include `ABS`, `ROUND`, `SQRT`, `LEN`, `LEFT`, `RIGHT`, `UPPER`, `DATE`, `YEAR`, `MONTH`, `NOW`, `ISNUMBER`, `TRUE`, `FALSE`, `AND`, `OR`, `NOT`, and `IFERROR`.

Functions that don't originate from Excel

Two useful DAX functions that aren't specific to modeling and that don't originate from Excel are `DISTINCTCOUNT` and `DIVIDE`.

DISTINCTCOUNT function

You can use the DISTINCTCOUNT DAX function to count the number of distinct values in a column. This function is especially powerful in an analytics solution. Consider that the count of customers is different from the count of *distinct* customers. The latter doesn't count repeat customers, so the difference is "How many customers" compared with "How many *different* customers."

DIVIDE function

You can use the DIVIDE DAX function to achieve division. You must pass in numerator and denominator expressions. Optionally, you can pass in a value that represents an *alternate result*. The `DIVIDE` function's syntax is:

DAX

```
DIVIDE(<numerator>, <denominator>[, <alternate_result>])
```

The `DIVIDE` function automatically handles division by zero cases. If an alternate result isn't passed in, and the denominator is zero or BLANK, the function returns BLANK. When an alternate result is passed in, it's returned instead of BLANK.

This function is convenient because it saves your expression from having to first test the denominator value. The function is also better optimized for testing the denominator value than the `IF` function. The performance gain is significant because checking for division by zero is expensive. What's more, using the `DIVIDE` function results in a more concise and elegant expression.

> **Tip**
>
> We recommend that you use the `DIVIDE` function whenever the denominator is an expression that could return zero or BLANK. In the case that the denominator is a constant value, we recommend that you use the divide operator (/), which is introduced later in this module. In this case, the division is guaranteed to succeed, and your expression will perform better because it will avoid unnecessary testing.

Next unit: Use DAX operators

Continue >

Use DAX operators

5 minutes

Your DAX formulas can use operators to create expressions that perform arithmetic calculations, compare values, work with strings, or test conditions.

> **Tip**
>
> Many DAX operators and precedence order are the same as those found in Excel.

Arithmetic operators

The following table lists the arithmetic operators.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation

Remember, when you are dividing two expressions, and when the denominator could return zero or BLANK, it's more efficient and safer to use the DIVIDE DAX function.

Comparison operators

The following table lists the comparison operators, which are used to compare two values. The result is either TRUE or FALSE.

Operator	Description
=	Equal to
==	Strict equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

All comparison operators, except **strict equal to** (==), treat BLANK as equal to the number zero, an empty string (""), the date December 30, 1899, or FALSE. It means that the expression `[Revenue] = 0` will be TRUE when the value of `[Revenue]` is either zero or BLANK. In contrast, `[Revenue] == 0` is TRUE only when the value of `[Revenue]` is zero.

Text concatenation operator

Use the ampersand (&) character to connect, or concatenate, two text values to produce one continuous text value. For example, consider the following calculated column definition:

DAX

```
Model Color = 'Product'[Model] & "–" & 'Product'[Color]
```

Logical operators

Use logical operators to combine expressions that produce a single result. The following table lists all logical operators.

Operator	Description
&&	Creates an AND condition between two expressions where each has a Boolean result. If both expressions return TRUE, the combination of the expressions also

Operator	Description
	returns TRUE; otherwise the combination returns FALSE.
|| (double pipe)	Creates an OR condition between two logical expressions. If either expression returns TRUE, the result is TRUE; only when both expressions are FALSE is the result FALSE.
IN	Creates a logical OR condition between each row that is being compared to a table. Note: The table constructor syntax uses braces.
NOT	Inverts the state of a Boolean expression (FALSE to TRUE, and vice versa).

An example that uses the `IN` logical operator is the **ANZ Revenue** measure definition, which uses the CALCULATE DAX function to enforce a specific filter of two countries: Australia and New Zealand.

> **Note**
>
> You'll be introduced to the powerful `CALCULATE` function when you learn how to modify the filter context.

DAX

```
ANZ Revenue =
CALCULATE(
    [Revenue],
    Customer[Country-Region] IN {
        "Australia",
        "New Zealand"
    }
)
```

Operator precedence

When your DAX formula includes multiple operators, DAX uses rules to determine the evaluation order, which is known as an *operator precedence*. Operations are ordered according to the following table.

Operator	Description
^	Exponentiation
-	Sign (as in -1)
* and /	Multiplication and division
NOT	NOT
+ and -	Addition and subtraction
&	Concatenation of two strings of text
=,==,<,>,<=,>=,<>	Comparison

When the operators have equal precedence value, they're ordered from left to right.

In general, operator precedence is the same as what's found in Excel. If you need to override the evaluation order, then group operations within parentheses.

For example, consider the following calculated column definition:

DAX

```
Extended Amount = Sales[Order Quantity] * Sales[Unit Price] * 1 - [Unit
Price Discount Pct]
```

This sample calculated column definition produces an incorrect result because multiplication happens before the subtraction. The following correct calculated column definition uses parentheses to ensure that the subtractions happen before the multiplications.

DAX

```
Extended Amount = Sales[Order Quantity] * Sales[Unit Price] * (1 - [Unit
Price Discount Pct])
```

Tip

Remembering operator precedence rules can be challenging, especially for DAX beginners. Consequently, we recommend that you test your formulas thoroughly. When

the formulas don't produce the correct result due to an incorrect evaluation order, you can experiment by adding parentheses to adjust the evaluation order. You can also add parentheses to improve the readability of your formulas.

For more information about DAX operators and precedence order, see DAX operators.

Implicit conversion

When writing a DAX formula that uses operators to combine different data types, you don't need to explicitly convert types. Usually, DAX automatically identifies the data types of referenced model objects and performs implicit conversions where necessary to complete the specified operation.

However, some limitations might exist on the values that can be successfully converted. If a value or a column has a data type that's incompatible with the current operation, DAX returns an error. For example, the attempt to multiply a date value will create an error because it isn't logical.

BLANK is handled differently, depending on the operator that is used. It's handled similar to how Excel treats BLANK, but differently to how databases (SQL) treat NULL. BLANK is treated as zero when acted on by arithmetic operators and as an empty string when concatenated to a string.

Tip

Remembering how BLANK is handled can be challenging, especially for DAX beginners. Consequently, we recommend that you test your formulas thoroughly. When BLANKs create unexpected results, consider using the **IF** and **ISBLANK** DAX functions to test for BLANK, and then respond in an appropriate way.

Next unit: Use DAX variables

Continue >

Use DAX variables

1 minute

You can declare DAX variables in your formula expressions. When you declare at least one variable, a `RETURN` clause is used to define the expression, which then refers to the variables.

We recommend that you use variables because they offer several benefits:

- Improving the readability and maintenance of your formulas.
- Improving performance because variables are evaluated once and only when or if they're needed.
- Allowing (at design time) straightforward testing of a complex formula by returning the variable of interest.

The following example shows a formula that declares a variable. The **Revenue YoY %** measure definition is rewritten to declare a variable that's assigned the value of the prior year's revenue.

DAX

```
Revenue YoY % =
VAR RevenuePriorYear =
    CALCULATE(
        [Revenue],
        SAMEPERIODLASTYEAR('Date'[Date])
    )
RETURN
    DIVIDE(
        [Revenue] - RevenuePriorYear,
        RevenuePriorYear
    )
```

Notice that the `RETURN` clause refers to the variable twice. This improved measure definition formula will run in at least half the time because it doesn't need to evaluate the prior year's revenue twice.

In the **Adventure Works DW 2020 M02.pbix** Power BI Desktop file, refactor the **Revenue YoY %** measure to produce the same result as the previous example.

For more information on using DAX variables, see Use variables to improve your formulas.

Next unit: Check your knowledge

Answer the following questions to see what you've learned.

1. You're using Power BI Desktop to develop a model. It has a table named Sales, which includes a column named CustomerKey. In reports, you need a calculation to show the number of different customers who have placed orders. What type of DAX calculation will you add to the model? *

- ○ Calculated table

  ✗ A calculated table will add a new table to the model. Also, a calculated table can't be added to the Sales table.

- ○ Calculated column
- ○ Computed column
- ○ Measure

  ✓ You can add a measure to the Sales table by using the DISTINCTCOUNT DAX function.

2. You're using Power BI Desktop to develop a model. It has a table named Customer, which includes a column named DateOfBirth. In reports, you need to group customers by current age. What type of DAX calculation will you add to the Customer table? *

- ○ Calculated table
- ○ Calculated column

  ✓ A calculated column can store the current age for each customer. This column could then be used in reports to group customers by age.

- ○ Computed column
- ○ Measure

3. You're using Power BI Desktop to develop a model. It has a table named Geography, which has two relationships to the Sales table. One relationship filters by customer region and the other filters by sales region. You need to create a role-playing dimension so that both filters are possible. What type of DAX calculation will you add to the model? *

Calculated table

✔ A calculated table could create a table that duplicates the Geography table data. It could then have an active relationship to the Sales table. Both geography tables would have active relationships to allow report users to filter by customer region or sales region.

Calculated column

Computed column

✘ A computed column would not allow multiple active relationships between the Geography and Sales tables. Also, it's not a DAX calculation.

Measure

4. You write a DAX formula that adds BLANK to the number 20. What will be the result? *

The result will be zero (0).

The result will be 20.

✔ BLANK is converted to zero when added to a number.

The result will be BLANK.

✘ BLANK is converted to zero when added to a number.

The result will be NULL.

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Introduction

7 minutes

You can write a Data Analysis Expressions (DAX) formula to add a *calculated table* to your model. The formula can duplicate or transform existing model data to produce a new table.

> **Note**
>
> A calculated table can't connect to external data; you must use Power Query to accomplish that task.

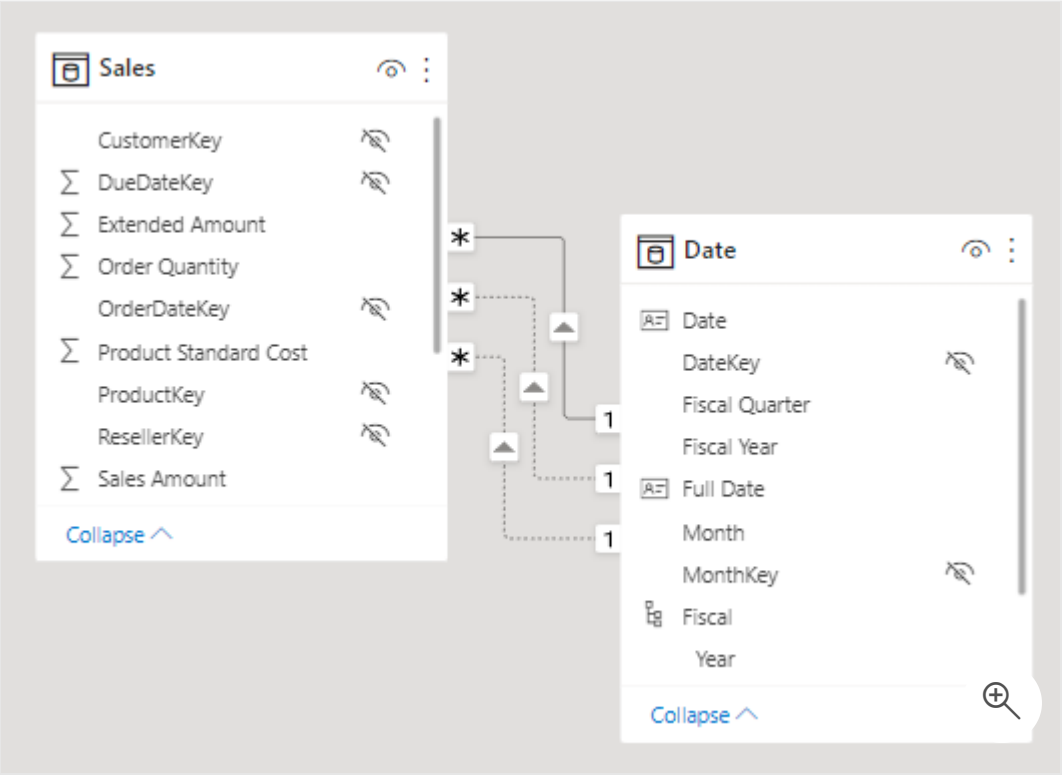
A calculated table formula must return a table object. The simplest formula can duplicate an existing model table.

Calculated tables have a cost: They increase the model storage size and they can prolong the data refresh time. The reason is because calculated tables recalculate when they have formula dependencies to refreshed tables.

Duplicate a table

The following section describes a common design challenge that can be solved by creating a calculated table. First, you should download and open the Adventure Works DW 2020 M03.pbix file and then switch to the model diagram.

In the model diagram, notice that the **Sales** table has three relationships to the **Date** table.



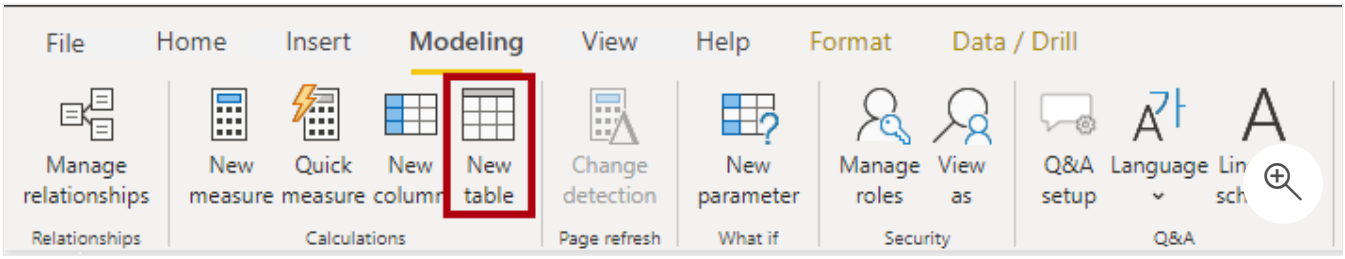
The model diagram shows three relationships because the Sales table stores sales data by order date, ship date, and due date. If you examine the **OrderDateKey**, **ShipDateKey**, and **DueDateKey** columns, notice that one relationship is represented by a solid line, which is the *active relationship*. The other relationships, which are represented by dashed lines, are *inactive relationships*.

> **Note**
>
> Only one active relationship can exist between any two model tables.

In the diagram, hover the cursor over the active relationship to highlight the related columns, which is how you would interact with the model diagram to learn about related columns. In this case, the active relationship filters the **OrderDateKey** column in the **Sales** table. Thus, filters that are applied to the **Date** table will propagate to the **Sales** table to filter by order date; they'll never filter by ship date or due date.

The next step is to delete the two inactive relationships between the **Date** table and the **Sales** table. To delete a relationship, right-click it and then select **Delete** in the context menu. Make sure that you delete both inactive relationships.

Next, add a new table to allow report users to filter sales by ship date. Switch to Report view and then, on the **Modeling** ribbon tab, from inside the **Calculations** group, select **New table**.

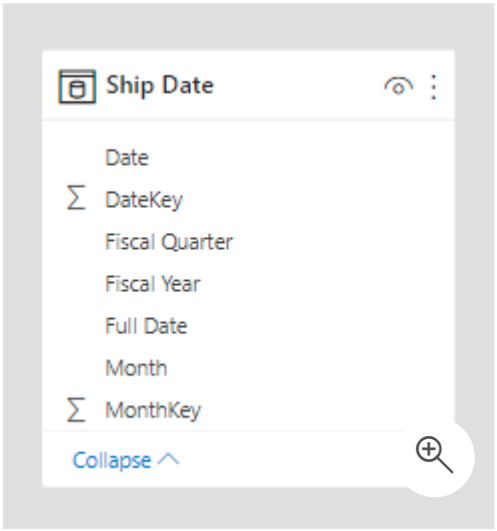


In the formula bar (located beneath the ribbon), enter the following calculated table definition and then press **Enter**.

```DAX
Ship Date = 'Date'
```

The calculated table definition duplicates the **Date** table data to produce a new table named **Ship Date**. The **Ship Date** table has exactly the same columns and rows as the **Date** table. When the **Date** table data refreshes, the **Ship Date** table recalculates, so they'll always be in sync.

Switch to the model diagram, and then notice the addition of the **Ship Date** table.



Next, create a relationship between the **DateKey** column in the **Ship Date** table and the **ShipDateKey** column in the **Sales** table. You can create the relationship by dragging the **DateKey** column in the **Ship Date** table onto the **ShipDateKey** column in the **Sales** table.

A calculated table only duplicates data; it doesn't duplicate any model properties or objects like column visibility or hierarchies. You'll need to set them up for the new table, if required.

> **Tip**
>
> It's possible to rename columns of a calculated table. In this example, it's a good idea to rename columns so that they better describe their purpose. For example, the **Fiscal Year**

column in the **Ship Date** table can be renamed as **Ship Fiscal Year**. Accordingly, when fields from the **Ship Date** table are used in visuals, their names are automatically included in captions like the visual title or axis labels.

To complete the design of the **Ship Date** table, you can:

- Rename the following columns:
  - **Date** as **Ship Date**
  - **Fiscal Year** as **Ship Fiscal Year**
  - **Fiscal Quarter** as **Ship Fiscal Quarter**
  - **Month** as **Ship Month**
  - **Full Date** as **Ship Full Date**
- Sort the **Ship Full Date** column by the **Ship Date** column.
- Sort the **Ship Month** column by the **MonthKey** column.
- Hide the **MonthKey** column.
- Create a hierarchy named **Fiscal** with the following levels:
  - Ship Fiscal Year
  - Ship Fiscal Quarter
  - Ship Month
  - Ship Full Date
- Mark the Ship Date table as a date table by using the **Ship Date** column.

Calculated tables are useful to work in scenarios when multiple relationships between two tables exist, as previously described. They can also be used to add a date table to your model. Date tables are required to apply special time filters known as *time intelligence*.

Create a date table

In the next example, a second calculated table will be created, this time by using the CALENDARAUTO DAX function.

Create the **Due Date** calculated table by using the following definition.

DAX

```
Due Date = CALENDARAUTO(6)
```

The `CALENDARAUTO` DAX function takes a single optional argument, which is the last month number of the year, and returns a single-column table. If you don't pass in a month number, it's assumed to be 12 (for December). For example, at Adventure Works, their financial year ends on June 30 of each year, so the value 6 (for June) is passed in.

The function scans all date and date/time columns in your model to determine the earliest and latest stored date values. It then produces a complete set of dates that span all dates in your model, ensuring that full years of dates are loaded. For example, if the earliest date that is stored in your model is October 15, 2021, then the first date that is returned by the `CALENDARAUTO` function would be July 1, 2021. If the latest date that is stored in the model is June 15, 2022, then the last date that is returned by the `CALENDARAUTO` function would be June 30, 2022.

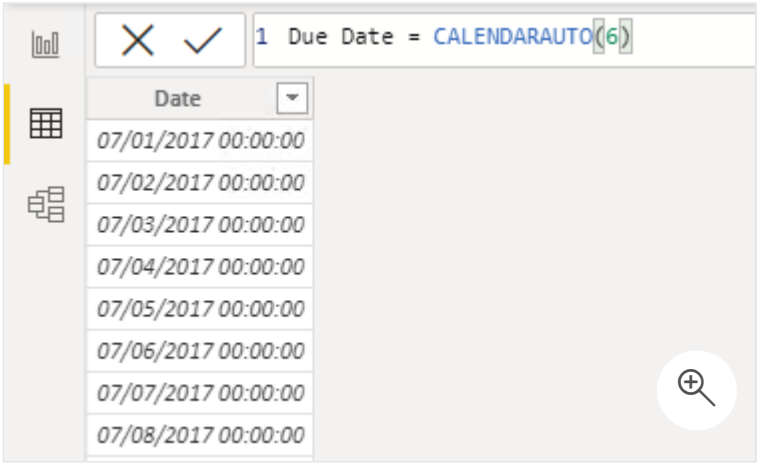
Effectively, the `CALENDARAUTO` function guarantees that the following requirements to *mark a date table* are met:

- The table must include a column of data type Date.
- The column must contain complete years.
- The column must not have missing dates.

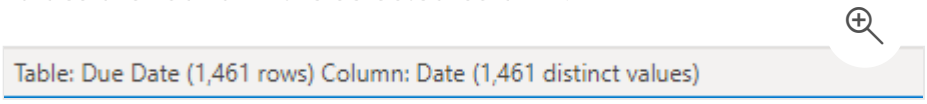
> **Tip**
>
> You can also create a date table by using the **CALENDAR** DAX function and passing in two date values, which represent the date range. The function generates one row for each date within the range. You can pass in static date values or pass in expressions that retrieve the earliest/latest dates from specific columns in your model.

Next, switch to data view, and then in the **Fields** pane, select the **Due Date** table. Now, review the column of dates. You might want to order them to see the earliest date in the first row by selecting the arrow inside the **Date** column header and then sorting in ascending order.

> **Note**
>
> Ordering or filtering columns doesn't change how the values are stored. These functions help you explore and understand the data.



Now that the **Date** column is selected, review the message in the status bar (located in the lower-left corner). It describes how many rows that the table stores and how many distinct values are found in the selected column.



When the table rows and distinct values are the same, it means that the column contains unique values. That factor is important for two reasons: It satisfies the requirements to mark a date table, and it allows this column to be used in a model relationship as the one-side.

The **Due Date** calculated table will recalculate each time a table that contains a date column refreshes. In other words, when a row is loaded into the **Sales** table with an order date of July 1, 2022, the **Due Date** table will automatically extend to include dates through to the end of the next year: June 30, 2023.

The **Due Date** table requires additional columns to support the known filtering and grouping requirements, specifically by year, quarter, and month.

Next unit: Create calculated columns

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Create calculated columns

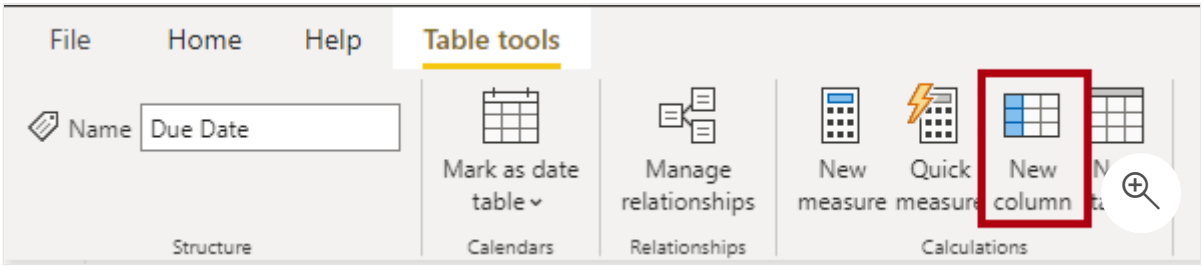
4 minutes

You can write a DAX formula to add a *calculated column* to any table in your model. A calculated column formula must return a scalar or single value.

Calculated columns in import models have a cost: They increase the model storage size and they can prolong the data refresh time. The reason is because calculated columns recalculate when they have formula dependencies to refreshed tables.

In data view, in the **Fields** pane, ensure that the **Due Date** table is selected. Before you create a calculated column, first rename the **Date** column to **Due Date**.

Now, you can add a calculated column to the **Due Date** table. To create a calculated column, in the **Table tools** contextual ribbon, from inside the **Calculations** group, select **New column**.



In the formula bar, enter the following calculated column definition and then press **Enter**.

DAX

```
Due Fiscal Year =
"FY"
    & YEAR('Due Date'[Due Date])
        + IF(
            MONTH('Due Date'[Due Date]) > 6,
            1
        )
```

The calculated column definition adds the **Due Fiscal Year** column to the **Due Date** table. The following steps describe how Microsoft Power BI evaluates the calculated column formula:

1. The addition operator (+) is evaluated before the text concatenation operator (&).
2. The YEAR DAX function returns the whole number value of the due date year.
3. The IF DAX function returns the value when the due date month number is 7-12 (July to December); otherwise, it returns BLANK. (For example, because the Adventure Works

financial year is July-June, the last six months of the calendar year will use the next calendar year as their financial year.)

4. The year value is added to the value that is returned by the `IF` function, which is the value one or BLANK. If the value is BLANK, it's implicitly converted to zero (0) to allow the addition to produce the fiscal year value.
5. The literal text value `"FY"` concatenated with the fiscal year value, which is implicitly converted to text.

Add a second calculated column by using the following definition:

DAX

```
Due Fiscal Quarter =
'Due Date'[Due Fiscal Year] & " Q"
    & IF(
        MONTH('Due Date'[Due Date]) <= 3,
        3,
        IF(
            MONTH('Due Date'[Due Date]) <= 6,
            4,
            IF(
                MONTH('Due Date'[Due Date]) <= 9,
                1,
                2
            )
        )
    )
```

The calculated column definition adds the **Due Fiscal Quarter** column to the **Due Date** table. The `IF` function returns the quarter number (Quarter 1 is July-September), and the result is concatenated to the **Due Fiscal Year** column value and the literal text `Q`.

Add a third calculated column by using the following definition:

DAX

```
Due Month =
FORMAT('Due Date'[Due Date], "yyyy mmm")
```

The calculated column definition adds the **Due Month** column to the **Due Date** table. The FORMAT DAX function converts the **Due Date** column value to text by using a format string. In this case, the format string produces a label that describes the year and abbreviated month name.

> **Note**

Many user-defined date/time formats exist. For more information, see **Custom date and time formats for the FORMAT function**.

Add a fourth calculated column by using the following definition:

DAX

```
Due Full Date =
FORMAT('Due Date'[Due Date], "yyyy mmm, dd")
```

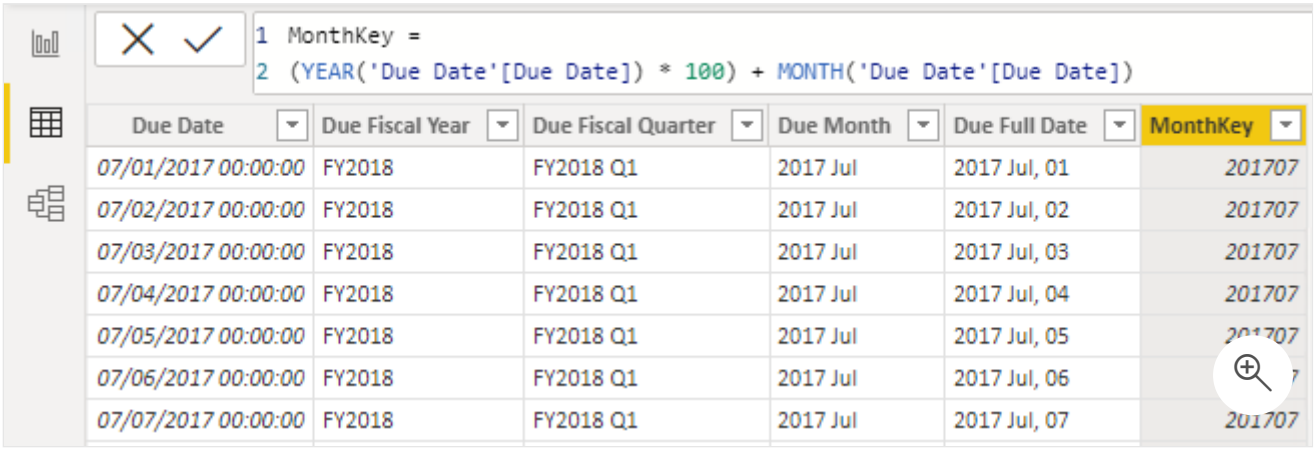
Add a fifth calculated column by using the following definition:

DAX

```
MonthKey =
(YEAR('Due Date'[Due Date]) * 100) + MONTH('Due Date'[Due Date])
```

The **MonthKey** calculated column multiplies the due date year by the value 100 and then adds the month number of the due date. It produces a numeric value that can be used to sort the **Due Month** text values in chronological order.

Verify that the **Due Date** table has six columns. The first column was added when the calculated table was created, and the other five columns were added as calculated columns.



Due Date	Due Fiscal Year	Due Fiscal Quarter	Due Month	Due Full Date	MonthKey
07/01/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 01	201707
07/02/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 02	201707
07/03/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 03	201707
07/04/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 04	201707
07/05/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 05	201707
07/06/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 06	
07/07/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 07	201707

To complete the design of the **Due Date** table, you can:

- Sort the **Due Full Date** column by the **Due Date** column.
- Sort the **Due Month** column by the **MonthKey** column.
- Hide the **MonthKey** column.
- Create a hierarchy named **Fiscal** with the following levels:
  - Due Fiscal Year
  - Due Fiscal Quarter
  - Due Month

    - Due Full Date
- Mark the **Due Date** table as a date table by using the **Due Date** column.

Next unit: Learn about row context

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Learn about row context

2 minutes

Now that you've created calculated columns, you can learn how their formulas are evaluated.

The formula for a calculated column is evaluated for each table row. Furthermore, it's evaluated within row context, which means *the current row*. Consider the **Due Fiscal Year** calculated column definition:

DAX

```
Due Fiscal Year =
"FY"
    & YEAR('Due Date'[Due Date])
        + IF(
            MONTH('Due Date'[Due Date]) <= 6,
            1
        )
```

When the formula is evaluated for each row, the `'Due Date'[Due Date]` column reference returns the column value for *that row*. You'll find that Microsoft Excel has the same concept for working with formulas in Excel tables .

However, row context doesn't extend beyond the table. If your formula needs to reference columns in other tables, you have two options:

- If the tables are related, directly or indirectly, you can use the RELATED or RELATEDTABLE DAX function. The `RELATED` function retrieves the value at the one-side of the relationship, while the `RELATEDTABLE` retrieves values on the many-side. The `RELATEDTABLE` function returns a table object.
- When the tables aren't related, you can use the LOOKUPVALUE DAX function.

Generally, try to use the `RELATED` function whenever possible. It will usually perform better than the `LOOKUPVALUE` function due to the ways that relationship and column data is stored and indexed.

Now, add the following calculated column definition to the **Sales** table:

DAX

```
Discount Amount =
(
    Sales[Order Quantity]
        * RELATED('Product'[List Price])
) - Sales[Sales Amount]
```

The calculated column definition adds the **Discount Amount** column to the **Sales** table. Power BI evaluates the calculated column formula for each row of the **Sales** table. The values for the **Order Quantity** and **Sales Amount** columns are retrieved within row context. However, because the **List Price** column belongs to the **Product** table, the `RELATED` function is required to retrieve the list price value *for the sale product*.

Row context is used when calculated column formulas are evaluated. It's also used when a class of functions, known as *iterator functions*, are used. Iterator functions provide you with flexibility to create sophisticated summarizations. Iterator functions are described in a later module.

Next unit: Choose a technique to add a column

Continue >

Choose a technique to add a column

2 minutes

There are three techniques that you can use to add columns to a model table:

- Add columns to a view or table (as a persisted column), and then source them in Power Query. This option only makes sense when your data source is a relational database and if you have the skills and permissions to do so. However, it's a good option because it supports ease of maintenance and allows reuse of the column logic in other models or reports.
- Add custom columns (using M) to Power Query queries.
- Add calculated columns (using DAX) to model tables.

Regardless of which technique you use, it results in the same outcome. Report users can't determine the origin of a column. Typically, they're not concerned with how the column was created but rather that it delivers the right data.

When multiple ways are available to add a column, you can consider using the approach that best fits your skills and that's supported by the language (M or DAX). However, the preference is to add custom columns in Power Query, whenever possible, because they load to the model in a more compact and optimal way.

When you need to add a column to a calculated table, make sure that you create a calculated column. Otherwise, we recommend that you only use a calculated column when the calculated column formula:

- Depends on summarized model data.
- Needs to use specialized modeling functions that are only available in DAX, such as the `RELATED` and `RELATEDTABLE` functions. Specialized functions can also include the DAX parent and child hierarchies, which are designed to naturalize a recursive relationship into columns, for example, in an employee table where each row stores a reference to the row of the manager (who is also an employee).

Next unit: Check your knowledge

200 XP

Check your knowledge

3 minutes

Answer the following questions to see what you've learned.

1. Which statement about calculated tables is true? *

- ( ) Calculated tables increase the size of the data model.
  - ✓ Calculated tables store data inside the model, and adding them results in a larger model size.
- ( ) Calculated tables are evaluated by using row context.
  - ✗ Calculated tables aren't evaluated in any context. Calculated columns are evaluated in row context.
- ( ) Calculated tables are created in Power Query.
- ( ) Calculated tables cannot include calculated columns.

2. Which statement about calculated columns is true? *

- ( ) Calculated columns are created in the Power Query Editor window.
- ( ) Calculated column formulas are evaluated by using row context.
  - ✓ Calculated column formulas are evaluated by using row context.
- ( ) Calculated column formulas can only reference columns from within their table.
  - ✗ Calculated column formulas can reference columns from other tables, but they need to use the RELATED or RELATEDTABLE function.
- ( ) Calculated columns can't be related to non-calculated columns.

3. You're developing a Power BI desktop model that sources data from an Excel workbook. The workbook has an employee table that stores one row for each employee. Each row has

a reference to the employee's manager, which is also a row in the employee table. You need to add several columns to the Employee table in your model to analyze payroll data within the organization hierarchy (like, executive level, manager level, and so on). Which technique will you use to add the columns? *

- ○ Add persisted columns to a table by using T-SQL.
- ○ Add column expressions in a view by using T-SQL.

  ✗ **The columns require naturalizing the parent-child relationship into columns. T-SQL can't be used to query the Excel workbook.**

- ○ Add computed columns by using M.
- ○ Add calculated columns by using DAX.

  ✓ **You can use the DAX parent-child functions to naturalize the recursive (employee-manager) relationship into columns.**

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Introduction

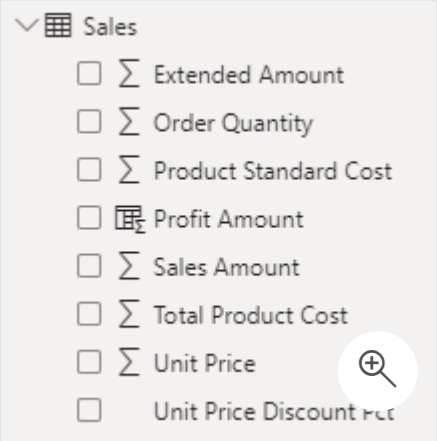
3 minutes

Measures in Microsoft Power BI models are either *implicit* or *explicit*. Implicit measures are automatic behaviors that allow visuals to summarize model column data. Explicit measures, also known simply as *measures*, are calculations that you can add to your model. This module focuses on how you can use implicit measures.

In the **Fields** pane, a column that's shown with the sigma symbol (∑) indicates two facts:

- It's a numeric column.
- It will summarize column values when it is used in a visual (when added to a field well that supports summarization).

In the following image, notice that the **Sales** table includes only fields that can be summarized, including the **Profit Amount** calculated column.



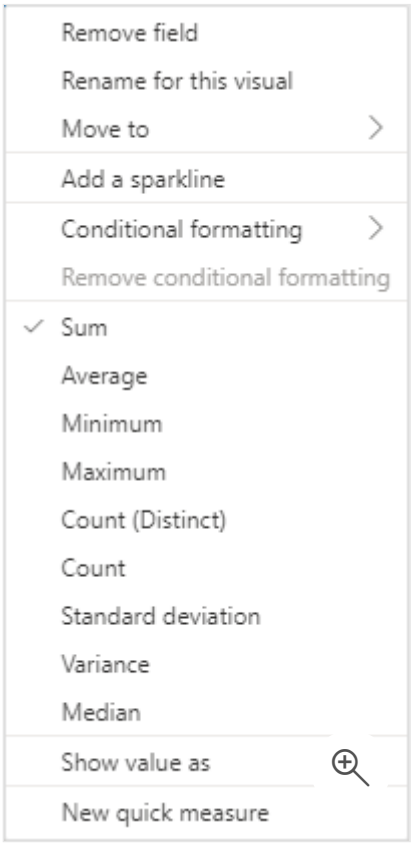
As a data modeler, you can control if and how the column summarizes by setting the **Summarization** property to **Don't summarize** or to a specific aggregation function. When you set the **Summarization** property to **Don't summarize**, the sigma symbol will no longer show next to the column in the **Fields** pane.

To observe how report authors can use implicit measures, you can first download and open the Adventure Works DW 2020 M04.pbix file.

In the report, from the **Sales** table, add the **Sales Amount** field to the matrix visual that groups fiscal year and month on its rows.

Fiscal Year	Sales Amount
FY2018	**$23,860,891.17**
2017 Jul	$1,423,357.32
2017 Aug	$2,057,902.45
2017 Sep	$2,523,947.55
2017 Oct	$561,6
2017 Nov	$4,764,9
2017 Dec	$596,746.56

To determine how the column is summarized, in the visual fields pane, for the **Sales Amount** field, select the arrow and then review the context menu options.

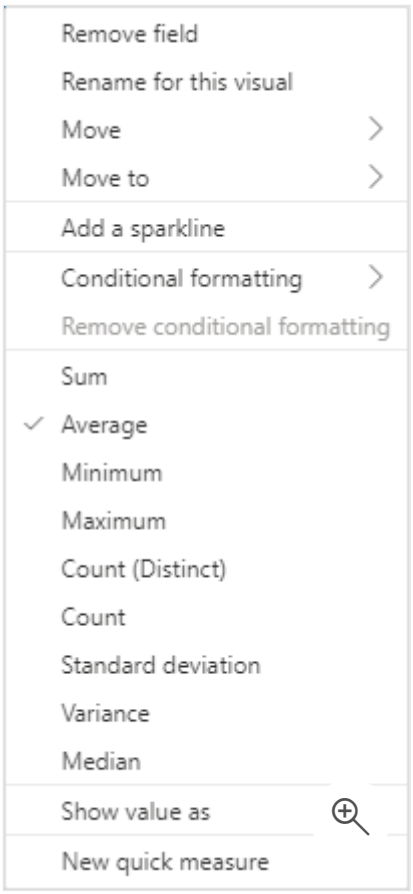


Notice that the **Sum** aggregation function has a check mark next to it. This check mark indicates that the column is summarized by *summing* column values together. It's also possible to change the aggregation function by selecting any of the other options like average, minimum, and so on.

Next, add the **Unit Price** field to the matrix visual.

Fiscal Year	Sales Amount	Unit Price
FY2018	**$23,860,891.17**	**$1,273.63**
2017 Jul	$1,423,357.32	$1,817.15
2017 Aug	$2,057,902.45	$1,181.42
2017 Sep	$2,523,947.55	$1,030.56
2017 Oct	$561,681.48	$3,
2017 Nov	$4,764,920.16	$1,
2017 Dec	$596,746.56	$3,174.18

The default summarization is now set to **Average** (the modeler knows that it's inappropriate to sum unit price values together because they're rates, which are non-additive).



Implicit measures allow the report author to start with a default summarization technique and lets them modify it to suit their visual requirements.

Numeric columns support the greatest range of aggregation functions:

- Sum
- Average
- Minimum
- Maximum
- Count (Distinct)
- Count
- Standard deviation
- Variance
- Median

Summarize non-numeric columns

Non-numeric columns can be summarized. However, the sigma symbol does not show next to non-numeric columns in the **Fields** pane because they don't summarize by default.

Text columns allow the following aggregations:

- First (alphabetically)
- Last (alphabetically)
- Count (Distinct)
- Count

Date columns allow the following aggregations:

- Earliest
- Latest
- Count (Distinct)
- Count

Boolean columns allow the following aggregations:

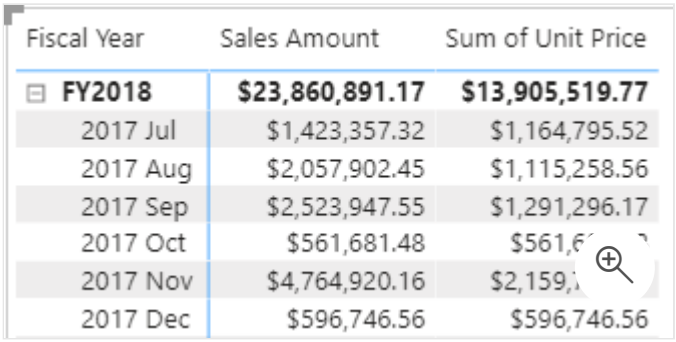
- Count (Distinct)
- Count

Benefits of implicit measures

Several benefits are associated with implicit measures. Implicit measures are simple concepts to learn and use, and they provide flexibility in the way that report authors visualize model data. Additionally, they mean less work for you as a data modeler because you don't have to create explicit calculations.

Limitations of implicit measures

Implicit measures do have limitations. Despite setting an appropriate summarization method, report authors could choose to aggregate a column in unsuitable ways. For example, in the matrix visual, you could modify the aggregate function of **Unit Price** to **Sum**.



Fiscal Year	Sales Amount	Sum of Unit Price
FY2018	**$23,860,891.17**	**$13,905,519.77**
2017 Jul	$1,423,357.32	$1,164,795.52
2017 Aug	$2,057,902.45	$1,115,258.56
2017 Sep	$2,523,947.55	$1,291,296.17
2017 Oct	$561,681.48	$561,6
2017 Nov	$4,764,920.16	$2,159,
2017 Dec	$596,746.56	$596,746.56

The report visual obeys your setup, but it has now produced a **Sum of Unit Price** column, which presents misleading data.

The most significant limitation of implicit measures is that they only work for simple scenarios, meaning that they can only summarize column values that use a specific aggregation function. Therefore, in situations when you need to calculate the ratio of each month's sales amount over the yearly sales amount, you'll need to produce an explicit measure by writing a Data Analysis Expressions (DAX) formula to achieve that more sophisticated requirement.

Implicit measures don't work when the model is queried by using Multidimensional Expressions (MDX). This language expects explicit measures and can't summarize column data. It's used when a Power BI dataset is queried by using Analyze in Excel or when a Power BI paginated report uses a query that is generated by the MDX graphical query designer.

Next unit: Create simple measures

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Create simple measures

3 minutes

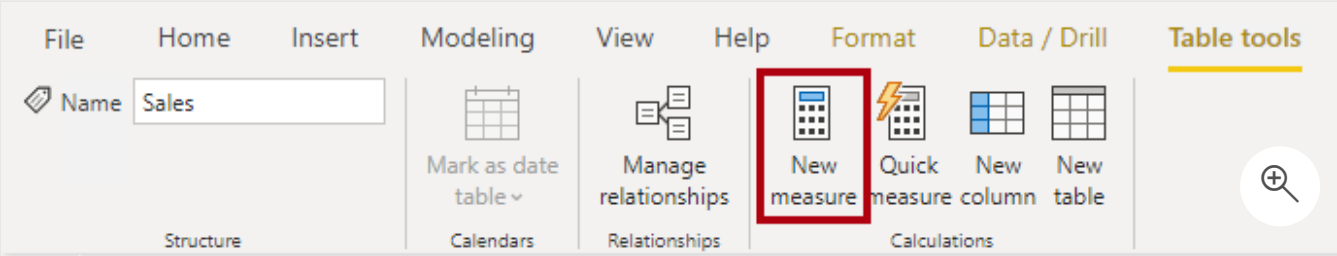
You can write a DAX formula to add a measure to any table in your model. A measure formula must return a scalar or single value.

> **Note**
>
> In tabular modeling, no such concept as a calculated measure exists. The word *calculated* is used to describe calculated tables and calculated columns. It distinguishes them from tables and columns that originate from Power Query, which doesn't have the concept of an explicit measure.

Measures don't store values in the model. Instead, they're used at query time to return summarizations of model data. Additionally, measures can't reference a table or column directly; they must pass the table or column into a function to produce a summarization.

A *simple* measure is one that aggregates the values of a single column; it does what implicit measures do automatically.

In the next example, you will add a measure to the **Sales** table. In the **Fields** pane, select the **Sales** table. To create a measure, in the **Table Tools** contextual ribbon, from inside the **Calculations** group, select **New measure**.



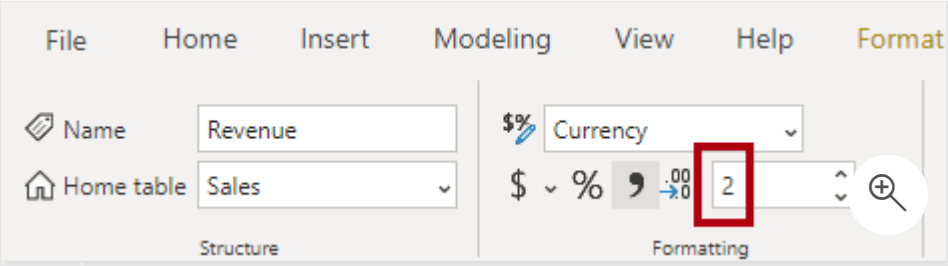
In the formula bar, enter the following measure definition and then press **Enter**.

DAX

```
Revenue =
SUM(Sales[Sales Amount])
```

The measure definition adds the **Revenue** measure to the **Sales** table. It uses the SUM DAX function to sum the values of the **Sales Amount** column.

On the **Measure tools** contextual ribbon, inside the **Formatting** group, set the decimal places to **2**.



> **Tip**
>
> Immediately after you create a measure, set the formatting options to ensure well-presented and consistent values in all report visuals.

Now, add the **Revenue** measure to the matrix visual. Notice that it produces the same result as the **Sales Amount** implicit measure.

In the matrix visual, remove **Sales Amount** and **Sum of Unit Price**.

Next, you will create more measures. Create the **Cost** measure by using the following measure definition, and then set the format with two decimal places.

DAX

```
Cost =
SUM(Sales[Total Product Cost])
```

Create the **Profit** measure, and then set the format with two decimal places.

DAX

```
Profit =
SUM(Sales[Profit Amount])
```

Notice that the **Profit Amount** column is a calculated column. This topic will be discussed later in this module.

Next, create the **Quantity** measure and format it as a whole number with the thousands separator.

DAX

```
Quantity =
```

```
SUM(Sales[Order Quantity])
```

Create three unit price measures and then set the format of each with two decimal places. Notice the different DAX aggregation functions that are used: MIN, MAX, and AVERAGE.

DAX

```
Minimum Price =
MIN(Sales[Unit Price])
```

DAX

```
Maximum Price =
MAX(Sales[Unit Price])
```

DAX

```
Average Price =
AVERAGE(Sales[Unit Price])
```

Now, hide the **Unit Price** column, which results in report authors losing their ability to summarize the column except by using your measures.

> **Tip**
>
> Adding measures and hiding columns is how you, the data modeler, can limit summarization options.

Next, create the following two measures, which count the number of orders and order lines. Format both measures with zero decimal places.

DAX

```
Order Line Count =
COUNT(Sales[SalesOrderLineKey])
```

DAX

```
Order Count =
DISTINCTCOUNT('Sales Order'[Sales Order])
```

The COUNT DAX function counts the number of non-BLANK values in a column, while the DISTINCTCOUNT DAX function counts the number of distinct values in a column. Because an order can have one or more order lines, the **Sales Order** column will have duplicate values. A distinct count of values in this column will correctly count the number of orders.

Alternatively, you can choose the better way to write the **Order Line Count** measure. Instead of counting values in a column, it's semantically clearer to use the COUNTROWS DAX function. Unlike the previously introduced aggregation functions, which aggregate column values, the `COUNTROWS` function counts the number of rows *for a table*.

Modify the **Order Line Count** measure formula you created above to the following parameters:

DAX

```
Order Line Count =
COUNTROWS(Sales)
```

Add each of the measures to the matrix visual.

All measures that you've created are considered simple measures because they aggregate a single column or single table.

Next unit: Create compound measures

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Create compound measures

1 minute

When a measure references one or more measures, it's known as a *compound measure*.

For this example, you will modify the **Profit** measure by using the following measure definition. Format the measure with two decimal places.

DAX

```
Profit =
[Revenue] - [Cost]
```

Next, add the **Profit** measure to the matrix visual.

Now that your model provides a way to summarize profit, you can delete the **Profit Amount** calculated column.

By removing this calculated column, you've optimized the data model. Removing this columns results in a decreased data model size and shorter data refresh times. The **Profit Amount** calculated column wasn't required because the **Profit** measure can directly produce the required result.

Next unit: Create quick measures

Continue

100 XP

Create quick measures

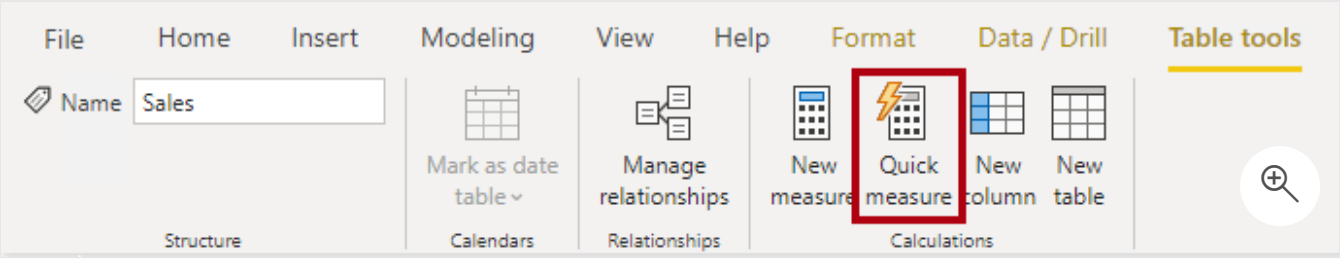
1 minute

Microsoft Power BI Desktop includes a feature named Quick Measures. This feature helps you to quickly perform common, powerful calculations by generating the DAX expression for you.

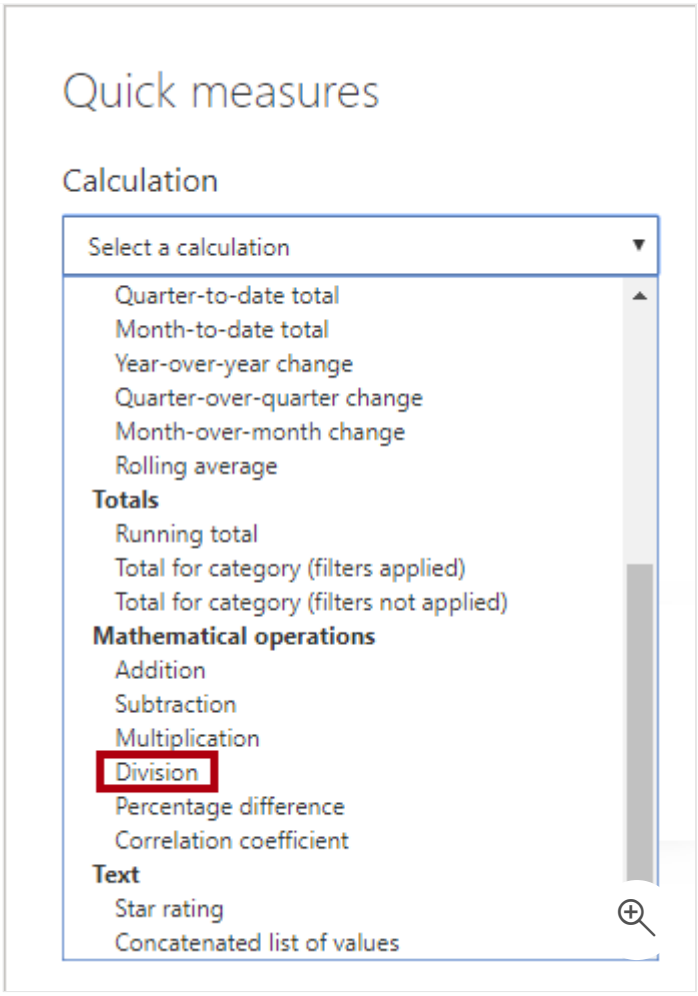
Many categories of calculations and ways to modify each calculation are available to fit your needs. Moreover, you are able to see the DAX that's generated by the quick measure and use it to jumpstart or expand your DAX knowledge.

In this next example, you'll create another compound measure to calculate profit margin. However, this time, you'll create it as a quick measure.

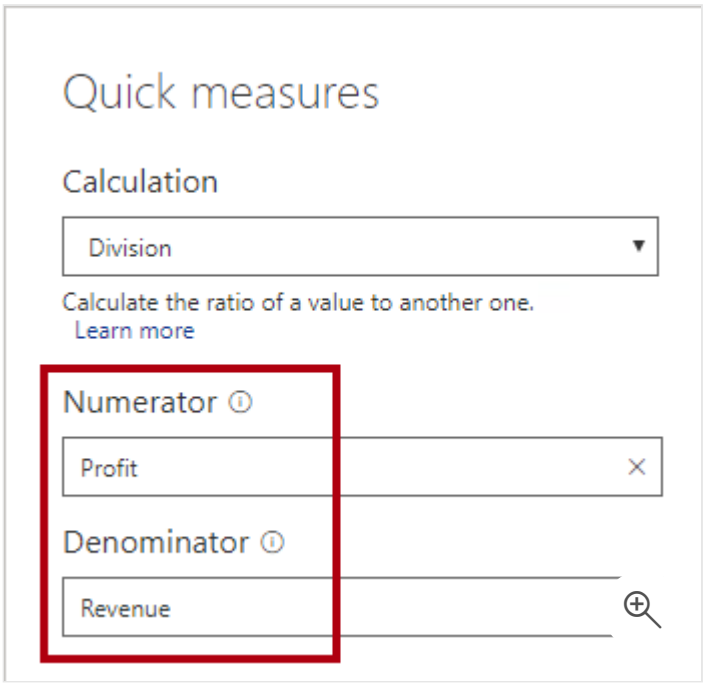
In the **Fields** pane, select the **Sales** table. On the **Table tools** contextual ribbon, from inside the **Calculations** group, select **Quick measure**.



In the **Quick measures** window, in the **Calculation** drop-down list, locate the **Mathematical operations** group (you might need to scroll down the list) and then select **Division**.



From the **Fields** list (in the **Quick measures** window), expand the **Sales** table and then drag the **Profit** measure into the **Numerator** box. Then, drag the **Revenue** measure into the **Denominator** box.



Select **OK**. In the **Fields** pane, notice the addition of the new compound measure. In the formula bar, review the measure definition.

DAX

```
Profit divided by Revenue =
DIVIDE([Profit], [Revenue])
```

> **Note**
>
> After the quick measure has been created, you must apply any changes in the formula bar.

Rename the measure as **Profit Margin**, and then set the format to a percentage with two decimal places.

Add the **Profit Margin** measure to the matrix visual.

Next unit: Compare calculated columns with measures

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Compare calculated columns with measures

1 minute

DAX beginners often experience a degree of confusion about calculated columns and measures. The following section reviews the similarities and differences between both.

Regarding similarities between calculated columns and measures, both are:

- Calculations that you can add to your data model.
- Defined by using a DAX formula.
- Referenced in DAX formulas by enclosing their names within square brackets.

The areas where calculated columns and measures differ include:

- **Purpose** - Calculated columns extend a table with a new column, while measures define how to summarize model data.
- **Evaluation** - Calculated columns are evaluated by using *row context* at data refresh time, while measures are evaluated by using *filter context* at query time. Filter context is introduced in a later module; it's an important topic to understand and master so that you can achieve complex summarizations.
- **Storage** - Calculated columns (in Import storage mode tables) store a value for each row in the table, but a measure never stores values in the model.
- **Visual use** - Calculated columns (like any column) can be used to filter, group, or summarize (as an implicit measure), whereas measures are designed to summarize.

Next unit: Check your knowledge

Continue

200 XP

Check your knowledge

3 minutes

Answer the following questions to see what you've learned.

1. Which statement about measures is correct? *

- Measures store values in the data model.
- Measures must be added to the data model to achieve summarization.
- Measures can reference columns directly.

✗ Measures cannot reference columns directly. They can only reference columns that have been passed into functions to summarize column values, like SUM.

- Measures can reference other measures directly.

✓ Measures can reference other measures. It's known as a compound measure.

2. Which DAX function can summarize a table? *

- SUM

✗ The SUM function can only summarize a column.

- AVERAGE
- COUNTROWS

✓ The COUNTROWS function summarizes a table by returning the number of rows.

- DISTINCTCOUNT

3. Which of the following statements describing similarity of measures and calculated columns in an Import model is true? *

- They can achieve summarization of model data.

✓ A calculated column can be summarized (implicit measure) and a measure always achieves summarization.

- They store values in the data model.
- They're evaluated during data refresh.
- They can be created by using quick calculations.

✗ While measures can be quickly created by using the Quick measures feature, no equivalent feature exists to create calculated columns.

Next unit: Exercise - Create DAX Calculations in Power BI Desktop

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Introduction

2 minutes

Watch the following video to learn about iterator functions.
https://www.microsoft.com/en-us/videoplayer/embed/RE4AkNG?postJsllMsg=true

Data Analysis Expressions (DAX) include a family of functions known as *iterator functions*. Iterator functions enumerate all rows of a given table and evaluate a given expression for each row. They provide you with flexibility and control over how your model calculations summarize data.

By now, you're familiar with single-column summarization functions, including `SUM`, `COUNT`, `MIN`, `MAX`, and others. Each of these functions has an equivalent iterator function that's identified by the "X" suffix, such as `SUMX`, `COUNTX`, `MINX`, `MAXX`, and others. Additionally, specialized iterator functions exist that perform filtering, ranking, semi-additive calculations over time, and more.

Characteristic of all iterator functions, you must pass in a table and an expression. The table can be a model table reference or an expression that returns a table object. The expression must evaluate to a scalar value.

Single-column summarization functions, like `SUM`, are shorthand functions. Internally, Microsoft Power BI converts the `SUM` function to `SUMX`. As a result, the following two measure definitions will produce the same result with the same performance.

DAX

```
Revenue = SUM(Sales[Sales Amount])
```

DAX

```
Revenue =
SUMX(
    Sales,
    Sales[Sales Amount]
)
```

It's important to understand how context works with iterator functions. Because iterator functions enumerate over table rows, the expression is evaluated for each row in row context, similar to calculated column formulas. The table is evaluated in filter context, so if you're using

the previous **Revenue** measure definition example, if a report visual was filtered by fiscal year **FY2020**, then the **Sales** table would contain sales rows that were ordered *in that year*. Filter context is described in the filter context module.

> **Important**
>
> When you're using iterator functions, make sure that you avoid using large tables (of rows) with expressions that use expansive DAX functions. Some functions, like the **SEARCH** DAX function, which scans a text value that looks for specific characters or text, can result in slow performance. Also, the **LOOKUPVALUE** DAX function might result in a slow, row-by-row retrieval of values. In this second case, use the **RELATED** DAX function instead, whenever possible.

Next unit: Use aggregation iterator functions

Continue >

100 XP

Use aggregation iterator functions

3 minutes

Each single-column summarization function has its equivalent iterator function. The following sections consider two aggregation scenarios when iterator functions are useful: complex summarization and higher grain summarization.

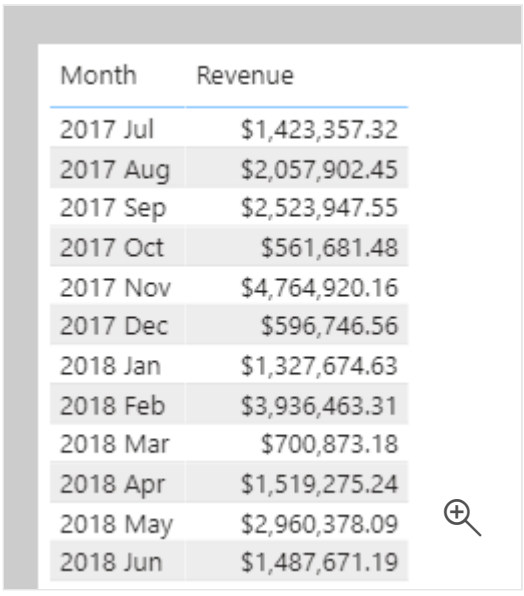
Complex summarization

In this section, you will create your first measure that uses an iterator function. First, download and open the Adventure Works DW 2020 M05.pbix file. Next, add the following measure definition:

DAX

```
Revenue =
SUMX(
    Sales,
    Sales[Order Quantity] * Sales[Unit Price] * (1 - Sales[Unit Price
Discount Pct])
)
```

Format the **Revenue** measure as currency with two decimal places, and then add it to the table visual that's found on **Page 1** of the report.



Month	Revenue
2017 Jul	$1,423,357.32
2017 Aug	$2,057,902.45
2017 Sep	$2,523,947.55
2017 Oct	$561,681.48
2017 Nov	$4,764,920.16
2017 Dec	$596,746.56
2018 Jan	$1,327,674.63
2018 Feb	$3,936,463.31
2018 Mar	$700,873.18
2018 Apr	$1,519,275.24
2018 May	$2,960,378.09
2018 Jun	$1,487,671.19

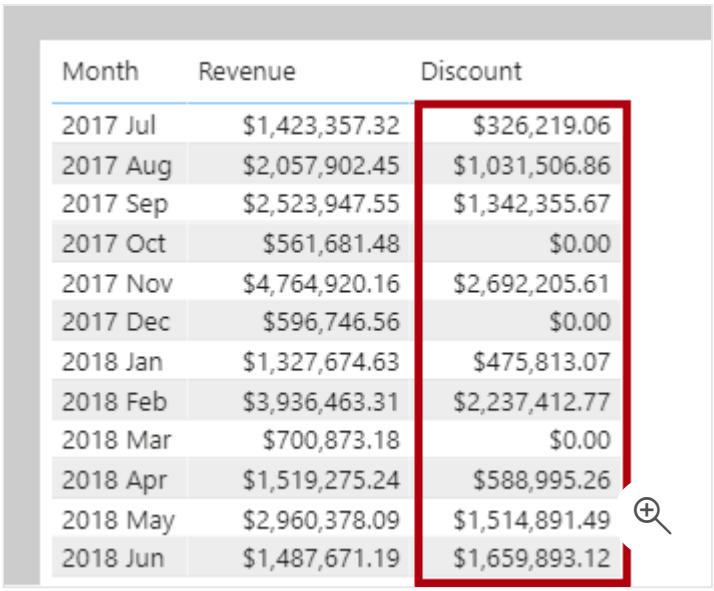
By using an iterator function, the **Revenue** measure formula aggregates more than the values of a single column. For each row, it uses the row context values of three columns to produce the revenue amount.

Now, add another measure:

DAX

```
Discount =
SUMX(
    Sales,
    Sales[Order Quantity]
    * (
        RELATED('Product'[List Price]) - Sales[Unit Price]
    )
)
```

Format the **Discount** measure as currency with two decimal places, and then add it to the table visual.



Month	Revenue	Discount
2017 Jul	$1,423,357.32	$326,219.06
2017 Aug	$2,057,902.45	$1,031,506.86
2017 Sep	$2,523,947.55	$1,342,355.67
2017 Oct	$561,681.48	$0.00
2017 Nov	$4,764,920.16	$2,692,205.61
2017 Dec	$596,746.56	$0.00
2018 Jan	$1,327,674.63	$475,813.07
2018 Feb	$3,936,463.31	$2,237,412.77
2018 Mar	$700,873.18	$0.00
2018 Apr	$1,519,275.24	$588,995.26
2018 May	$2,960,378.09	$1,514,891.49
2018 Jun	$1,487,671.19	$1,659,893.12

Notice that the formula uses the `RELATED` function. Remember, row context does not extend beyond the table. If your formula needs to reference columns in other tables, and model relationships exist between the tables, use the `RELATED` function for the one-side relationship or the `RELATEDTABLE` function for the many-side relationship.

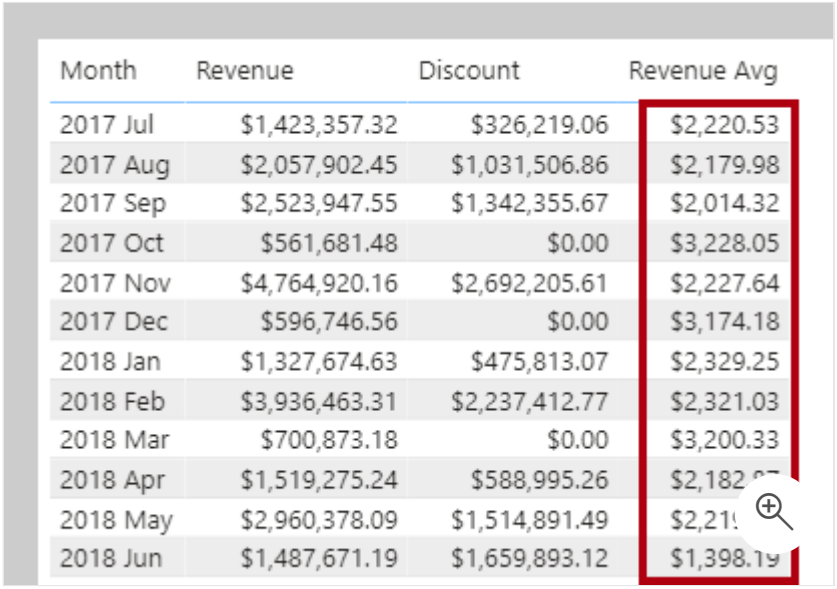
Higher grain summarization

The following example considers a requirement to report on average revenue. Add the following measure:

DAX

```
Revenue Avg =
AVERAGEX(
    Sales,
    Sales[Order Quantity] * Sales[Unit Price] * (1 - Sales[Unit Price
Discount Pct])
)
```

Format the **Revenue Avg** measure as currency with two decimal places, and then add it to the table visual.



Month	Revenue	Discount	Revenue Avg
2017 Jul	$1,423,357.32	$326,219.06	$2,220.53
2017 Aug	$2,057,902.45	$1,031,506.86	$2,179.98
2017 Sep	$2,523,947.55	$1,342,355.67	$2,014.32
2017 Oct	$561,681.48	$0.00	$3,228.05
2017 Nov	$4,764,920.16	$2,692,205.61	$2,227.64
2017 Dec	$596,746.56	$0.00	$3,174.18
2018 Jan	$1,327,674.63	$475,813.07	$2,329.25
2018 Feb	$3,936,463.31	$2,237,412.77	$2,321.03
2018 Mar	$700,873.18	$0.00	$3,200.33
2018 Apr	$1,519,275.24	$588,995.26	$2,182
2018 May	$2,960,378.09	$1,514,891.49	$2,21
2018 Jun	$1,487,671.19	$1,659,893.12	$1,398.19

Consider that *average* means the sum of values divided by the count of values. However, that theory raises a question: What does the count of values represent? In this case, the count of values is the number of expressions that didn't evaluate to BLANK. Also, because the iterator function enumerates the **Sales** table rows, average would mean *revenue per row*. Taking this logic one step further, because each row in the **Sales** table records a sales order line, it can be more precisely described as *revenue per order line*.

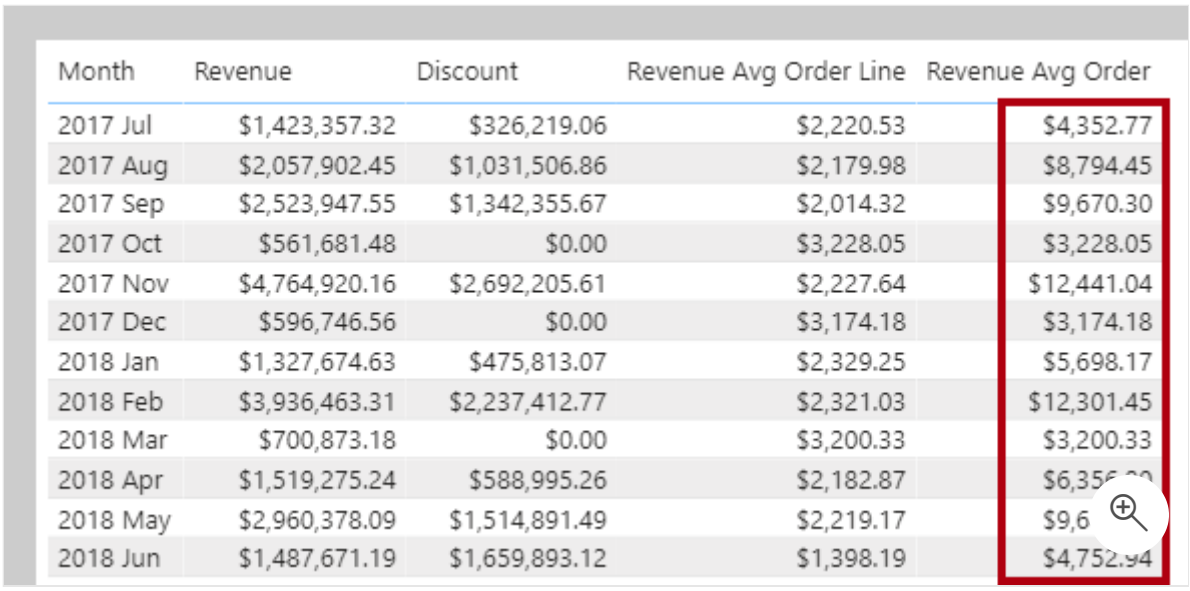
Accordingly, you should rename the **Revenue Avg** measure as **Revenue Avg Order Line** so that it's clear to report users about what's being used as the average base.

The following example uses an iterator function to create a new measure that raises the granularity to the sales order level (a sales order consists of one or more order lines). Add the following measure:

DAX

```
Revenue Avg Order =
AVERAGEX(
    VALUES('Sales Order'[Sales Order]),
    [Revenue]
)
```

Format the **Revenue Avg Order** measure as currency with two decimal places, and then add it to the table visual.



Month	Revenue	Discount	Revenue Avg Order Line	Revenue Avg Order
2017 Jul	$1,423,357.32	$326,219.06	$2,220.53	$4,352.77
2017 Aug	$2,057,902.45	$1,031,506.86	$2,179.98	$8,794.45
2017 Sep	$2,523,947.55	$1,342,355.67	$2,014.32	$9,670.30
2017 Oct	$561,681.48	$0.00	$3,228.05	$3,228.05
2017 Nov	$4,764,920.16	$2,692,205.61	$2,227.64	$12,441.04
2017 Dec	$596,746.56	$0.00	$3,174.18	$3,174.18
2018 Jan	$1,327,674.63	$475,813.07	$2,329.25	$5,698.17
2018 Feb	$3,936,463.31	$2,237,412.77	$2,321.03	$12,301.45
2018 Mar	$700,873.18	$0.00	$3,200.33	$3,200.33
2018 Apr	$1,519,275.24	$588,995.26	$2,182.87	$6,35
2018 May	$2,960,378.09	$1,514,891.49	$2,219.17	$9,6
2018 Jun	$1,487,671.19	$1,659,893.12	$1,398.19	$4,752.94

As expected, the average revenue for an order is always higher than the average revenue for a single order line.

Notice that the formula uses the VALUES DAX function. This function lets your formulas determine what values are in filter context. In this case, the `AVERAGEX` function iterates over each sales order *in filter context*. In other words, it iterates over each sales order for the month. Filter context and the `VALUES` function are introduced in the filter context module.

Next unit: Calculate ranks

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Calculate ranks

3 minutes

The RANKX DAX function is a special iterator function you can use to calculate ranks. Its syntax is as follows:

DAX

```
RANKX(<table>, <expression>[, <value>[, <order>[, <ties>]]])
```

Similar to all iterator functions, you must pass in a table and an expression. Optionally, you can pass in a rank value, set the order direction, or determine how to handle ranks when values are tied.

Order direction

Order direction is either ascending or descending. When ranking something favorable, like revenue values, you're likely to use descending order so that the highest revenue is ranked first. When ranking something unfavorable, like customer complaints, you might use ascending order so that the lowest number of complaints is ranked first. When you don't pass in an order argument, the function will use **0 (zero)** (for descending order).

Handle ties

You can handle ties by skipping rank values or using dense ranking, which uses the next rank value after a tie. When you don't pass in a ties argument, the function will use **Skip**. You'll have an opportunity to work with an example of each tie argument later in this unit.

Create ranking measures

Add the following measure to the **Product** table:

DAX

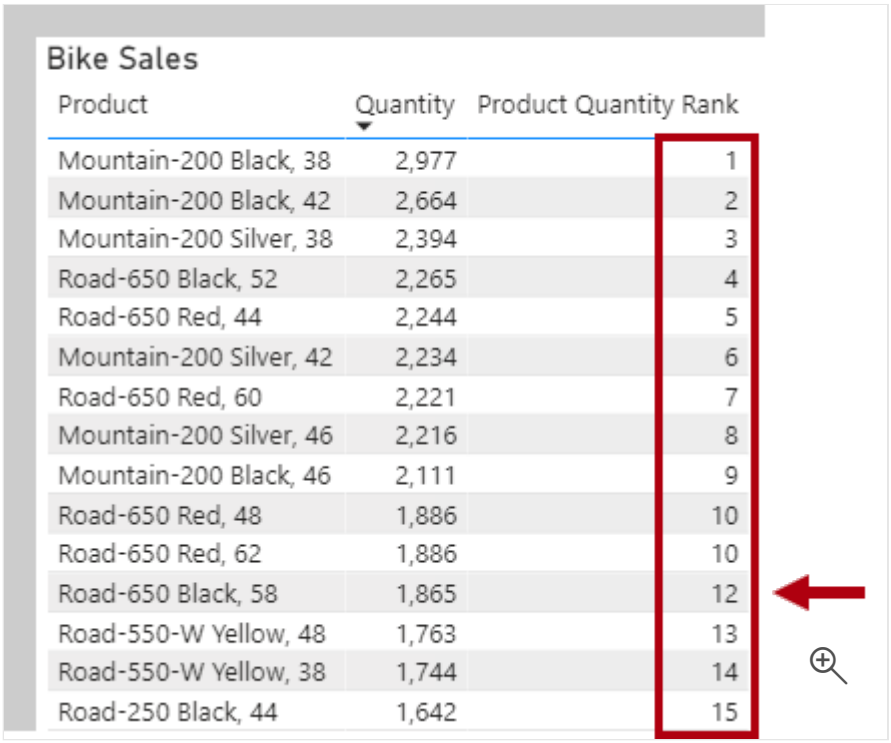
```
Product Quantity Rank =
RANKX(
    ALL('Product'[Product]),
```

```
    [Quantity]
)
```

Add the **Product Quantity Rank** measure to the table visual that is found on **Page 2** of the report. The table visual groups bike products and displays quantity, which orders products by descending quantity.

The `RANKX` function iterates over a table that is returned by the ALL DAX function. The `ALL` function is used to return all rows in a model table or values in one or more columns, and it *ignores all filters*. Therefore, in this case, it returns a table that consists of all **Product** column values in the **Product** table. The `RANKX` function must use the `ALL` function because the table visual will group by product (which is a filter on the **Product** table).

In the table visual, notice that two products tie for tenth place and that the next product's rank is 12. This visual is an example of using the **Skipped** ties argument.



Product	Quantity	Product Quantity Rank
Mountain-200 Black, 38	2,977	1
Mountain-200 Black, 42	2,664	2
Mountain-200 Silver, 38	2,394	3
Road-650 Black, 52	2,265	4
Road-650 Red, 44	2,244	5
Mountain-200 Silver, 42	2,234	6
Road-650 Red, 60	2,221	7
Mountain-200 Silver, 46	2,216	8
Mountain-200 Black, 46	2,111	9
Road-650 Red, 48	1,886	10
Road-650 Red, 62	1,886	10
Road-650 Black, 58	1,865	12
Road-550-W Yellow, 48	1,763	13
Road-550-W Yellow, 38	1,744	14
Road-250 Black, 44	1,642	15

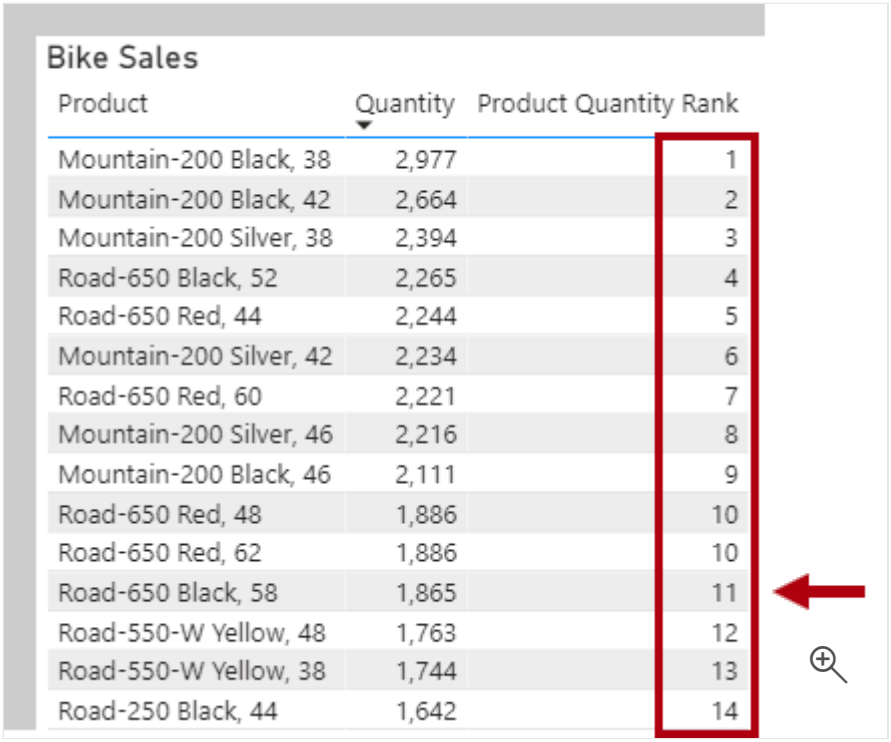
Your next task is to enter the following logic to modify the **Product Quantity Rank** measure definition to use dense ranking:

DAX

```
Product Quantity Rank =
RANKX(
    ALL('Product'[Product]),
    [Quantity],
    ,
    ,
```

```
    DENSE
)
```

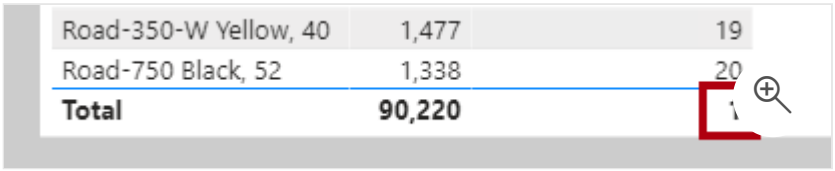
In the table visual, notice that a skipped ranking no longer exists. After the two products that tie for tenth place, the next ranking is 11.



Bike Sales

Product	Quantity	Product Quantity Rank
Mountain-200 Black, 38	2,977	1
Mountain-200 Black, 42	2,664	2
Mountain-200 Silver, 38	2,394	3
Road-650 Black, 52	2,265	4
Road-650 Red, 44	2,244	5
Mountain-200 Silver, 42	2,234	6
Road-650 Red, 60	2,221	7
Mountain-200 Silver, 46	2,216	8
Mountain-200 Black, 46	2,111	9
Road-650 Red, 48	1,886	10
Road-650 Red, 62	1,886	10
Road-650 Black, 58	1,865	11
Road-550-W Yellow, 48	1,763	12
Road-550-W Yellow, 38	1,744	13
Road-250 Black, 44	1,642	14

Notice that the table visual total for the **Product Quantity Rank** is one (1). The reason is because the total for all products is ranked.

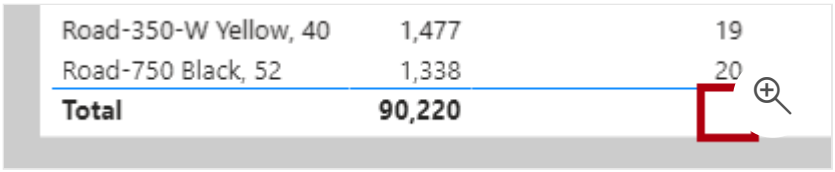


Road-350-W Yellow, 40	1,477	19
Road-750 Black, 52	1,338	20
Total	**90,220**	1

It's not appropriate to rank total products, so you will now use the following logic to modify the measure definition to return BLANK, unless a single product is filtered:

DAX

```
Product Quantity Rank =
IF(
    HASONEVALUE('Product'[Product]),
    RANKX(
        ALL('Product'[Product]),
        [Quantity],
        ,
        ,
        DENSE
    )
)
```



Notice that the total **Product Quantity Rank** is now BLANK, which was achieved by using the HASONEVALUE DAX function to test whether the **Product** column in the **Product** table has a single value in filter context. It's the case for each product group, but not for the total, which represents all products.

Filter context and the `HASONEVALUE` function will be introduced in the filter context module.

Next unit: Check your knowledge

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

200 XP

Check your knowledge

3 minutes

Answer the following questions to see what you've learned.

1. An iterator function always includes at least two arguments. What are they? *

- Column and expression
- Table and expression

  ✔ An iterator function iterates over a table and evaluates an expression for each row of the table.
- Measure and expression

  ✘ An iterator function iterates over a table and evaluates an expression for each row of the table. A measure is an expression.
- Table and ranking

2. Which statement about DAX iterator functions is true? *

- When used to add values of the same column, the SUM function performs better than the SUMX function.
- When used to add values of the same column, the SUMX function performs better than the SUM function.

  ✘ The SUM function and SUMX function perform the same. Internally, Power BI translates the SUM function as if it were written with the SUMX function.
- Iterator functions iterate over tables and evaluate an expression for each table row.

  ✔ Iterator functions iterate over tables and evaluate an expression for each table row.

- ◯ Iterator functions iterate over expressions and evaluate tables for each row.

3. You're developing a Power BI Desktop model. You need to create a measure formula by using the RANKX DAX function to rank students by their test results. The lowest rank should be assigned to the highest test result. Also, if multiple students achieve the same rank, the next student rank should follow on from the tied rank number. Which order and ties arguments should you pass to the RANKX function? *

- ◯ Order Ascending | Ties Skip
- ◯ Order Ascending | Ties Dense
- ◯ Order Descending | Ties Skip

  ✗ **The highest test result should be assigned the lowest rank. Therefore, the order argument must be Descending. Because no missing rank numbers should exist, the ties argument must be Dense.**

- ◯ Order Descending | Ties Dense

  ✓ **The highest test result should be assigned the lowest rank. Therefore, the order argument must be Descending. Because no missing rank numbers should exist, the ties argument must be Dense.**

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

Introduction

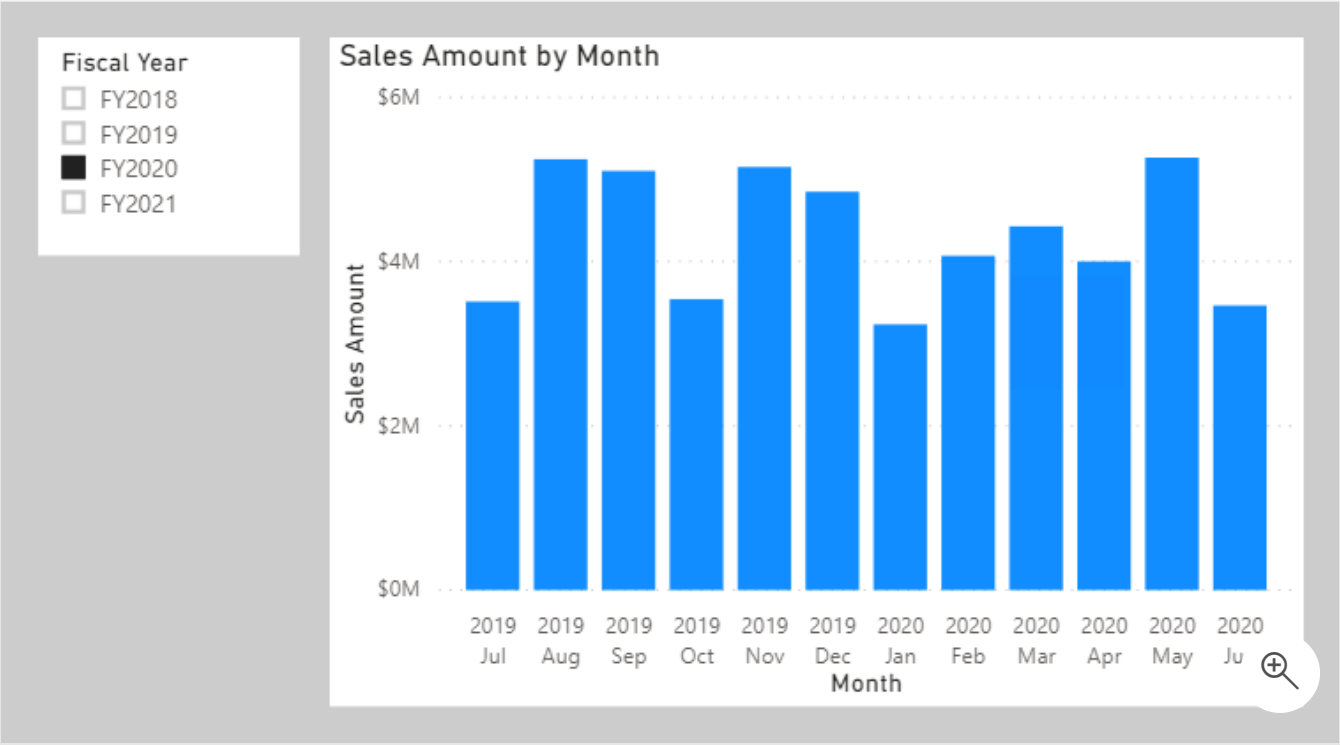
5 minutes

Watch the following video to learn about filter context.
https://www.microsoft.com/en-us/videoplayer/embed/RE4AvVc?postJsllMsg=true

Filter context describes the filters that are applied during the evaluation of a measure or measure expression. Filters can be applied directly to columns, like a filter on the **Fiscal Year** column in the **Date** table for the value **FY2020**. Additionally, filters can be applied indirectly, which happens when model relationships propagate filters to other tables. For example, the **Sales** table receives a filter through its relationship with the **Date** table, filtering the **Sales** table rows to those with an **OrderDateKey** column value in **FY2020**.

> **Note**
>
> Calculated tables and calculated columns aren't evaluated within filter context. Calculated columns are evaluated in row context, though the formula can transition the row context to filter context, if it needs to summarize model data. Context transition is described in Unit 5.

At report design time, filters are applied in the **Filters** pane or to report visuals. The slicer visual is an example of a visual whose only purpose is to filter the report page (and other pages when it's configured as a synced slicer). Report visuals, which perform grouping, also apply filters. They're implied filters; the difference is that the filter result is visible in the visual. For example, a stacked column chart visual can filter by fiscal year FY2020, group by month, and summarize sales amount. The fiscal year filter isn't visible in the visual result, yet the grouping, which results in a column for each month, behaves as a filter.



Not all filters are applied at report design time. Filters can be added when a report user interacts with the report. They can modify filter settings in the **Filters** pane, and they can cross-filter or cross-highlight visuals by selecting visual elements like columns, bars, or pie chart segments. These interactions apply additional filters to report page visuals (unless interactions have been disabled).

It's important to understand how filter context works. It guides you in defining the correct formula for your calculations. As you write more complex formulas, you'll identify times when you need to add, modify, or remove filters to achieve the desired result.

Consider an example that requires your formula to modify the filter context. Your objective is to produce a report visual that shows each sales region together with its revenue and revenue *as a percentage of total revenue*.

Region	Revenue	Revenue % Total Region
Australia	$10,655,335.96	9.70%
Canada	$16,355,770.46	14.89%
Central	$7,909,009.01	7.20%
France	$7,251,555.65	6.60%
Germany	$4,878,300.38	4.44%
Northeast	$6,939,374.48	6.32%
Northwest	$16,084,942.55	14.65%
Southeast	$7,879,655.07	7.18%
Southwest	$24,184,609.60	22.02%
United Kingdom	$7,670,721.04	6.99%
Total	**$109,809,274.20**	**100.0(**

The **Revenue % Total Region** result is achieved by defining a measure expression that's the ratio of revenue divided by revenue *for all regions*. Therefore, for Australia, the ratio is 10,655,335.96 dollars divided by 109,809,274.20 dollars, which is 9.7 percent.

The numerator expression doesn't need to modify filter context; it should use the current filter context (a visual that groups by region applies a filter for that region). The denominator expression, however, needs to remove any region filters to achieve the result for all regions.

> **Tip**
>
> The key to writing complex measures is mastering these concepts:
>
> - Understanding how filter context works.
> - Understanding when and how to modify or remove filters to achieve a required result.
> - Composing a formula to accurately and efficiently modify filter context.
>
> Mastering these concepts takes practice and time. Rarely will students understand the concepts from the beginning of training. Therefore, be patient and persevere with the theory and activities. We recommend that you repeat this module at a later time to help reinforce key lessons.

The next unit introduces the CALCULATE DAX function. It's one of the most powerful DAX functions, allowing you to modify filter context when your formulas are evaluated.

Next unit: Modify filter context

Continue >

100 XP

Modify filter context

7 minutes

You can use the CALCULATE DAX function to modify filter context in your formulas. The syntax for the CALCULATE function is as follows:

```DAX
CALCULATE(<expression>, [[<filter1>], <filter2>]…)
```

The function requires passing in an expression that returns a scalar value and as many filters as you need. The expression can be a measure (which is a named expression) or any expression that can be evaluated in filter context.

Filters can be Boolean expressions or table expressions. It's also possible to pass in filter modification functions that provide additional control when you're modifying filter context.

When you have multiple filters, they're evaluated by using the `AND` logical operator, which means that all conditions must be `TRUE` at the same time.

> **Note**
>
> The **CALCULATETABLE** DAX function performs exactly the same functionality as the `CALCULATE` function, except that it modifies the filter context that's applied to an expression that returns a table object. In this module, the explanations and examples use the `CALCULATE` function, but keep in mind that these scenarios could also apply to the `CALCULATETABLE` function.

Apply Boolean expression filters

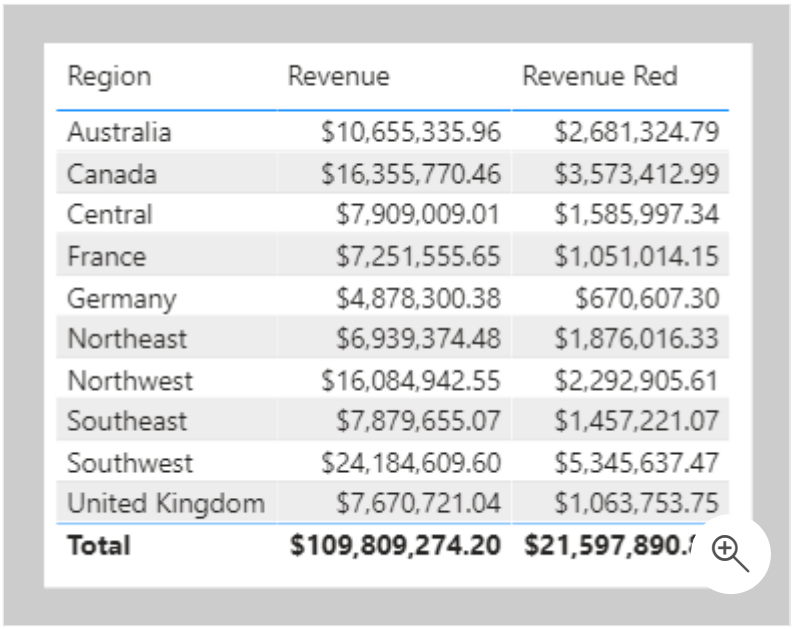
A Boolean expression filter is an expression that evaluates to `TRUE` or `FALSE`. Boolean filters must abide by the following rules:

- They can reference only a single column.
- They cannot reference measures.
- They cannot use functions that scan or return a table that includes aggregation functions like `SUM`.

In this example, you will create a measure. First, download and open the Adventure Works DW 2020 M06.pbix file. Then add the following measure to the **Sales** table that filters the **Revenue** measure by using a Boolean expression filter for red products.

```
DAX
Revenue Red = CALCULATE([Revenue], 'Product'[Color] = "Red")
```

Add the **Revenue Red** measure to the table visual that is found on **Page 1** of the report.



Region	Revenue	Revenue Red
Australia	$10,655,335.96	$2,681,324.79
Canada	$16,355,770.46	$3,573,412.99
Central	$7,909,009.01	$1,585,997.34
France	$7,251,555.65	$1,051,014.15
Germany	$4,878,300.38	$670,607.30
Northeast	$6,939,374.48	$1,876,016.33
Northwest	$16,084,942.55	$2,292,905.61
Southeast	$7,879,655.07	$1,457,221.07
Southwest	$24,184,609.60	$5,345,637.47
United Kingdom	$7,670,721.04	$1,063,753.75
Total	**$109,809,274.20**	**$21,597,890.**

In this next example, the following measure filters the **Revenue** measure by multiple colors. Notice the use of the `IN` operator followed by a list of color values.

```
DAX
Revenue Red or Blue = CALCULATE([Revenue], 'Product'[Color] IN {"Red",
"Blue"})
```

The following measure filters the **Revenue** measure by expensive products. Expensive products are those with a list price greater than USD 1000.

```
DAX
Revenue Expensive Products = CALCULATE([Revenue], 'Product'[List Price] >
1000)
```

Apply table expression filters

A table expression filter applies a table object as a filter. It could be a reference to a model table; however, it's likely a DAX function that returns a table object.

Commonly, you'll use the FILTER DAX function to apply complex filter conditions, including those that can't be defined by a Boolean filter expression. The `FILTER` function is classed as an iterator function, and so you would pass in a table, or table expression, and an expression to evaluate for each row of that table.

The `FILTER` function returns a table object with exactly the same structure as one that the table passed in. Its rows are a subset of those rows that were passed in, meaning the rows where the expression evaluated as `TRUE`.

The following example shows a table filter expression that uses the `FILTER` function:

DAX

```
Revenue High Margin Products =
CALCULATE(
    [Revenue],
    FILTER(
        'Product',
        'Product'[List Price] > 'Product'[Standard Cost] * 2
    )
)
```

In this example, the `FILTER` function filters all rows of the Product table that are in filter context. Each row for a product where its list price exceeds double its standard cost is displayed as a row of the filtered table. Therefore, the **Revenue** measure is evaluated for all products that are returned by the `FILTER` function.

All filter expressions that are passed in to the `CALCULATE` function are table filter expressions. A Boolean filter expression is a shorthand notation to improve the writing and reading experience. Internally, Microsoft Power BI translates Boolean filter expressions to table filter expressions, which is how it translates your **Revenue Red** measure definition.

DAX

```
Revenue Red =
CALCULATE(
    [Revenue],
    FILTER(
        'Product',
        'Product'[Color] = "Red"
    )
)
```

Filter behavior

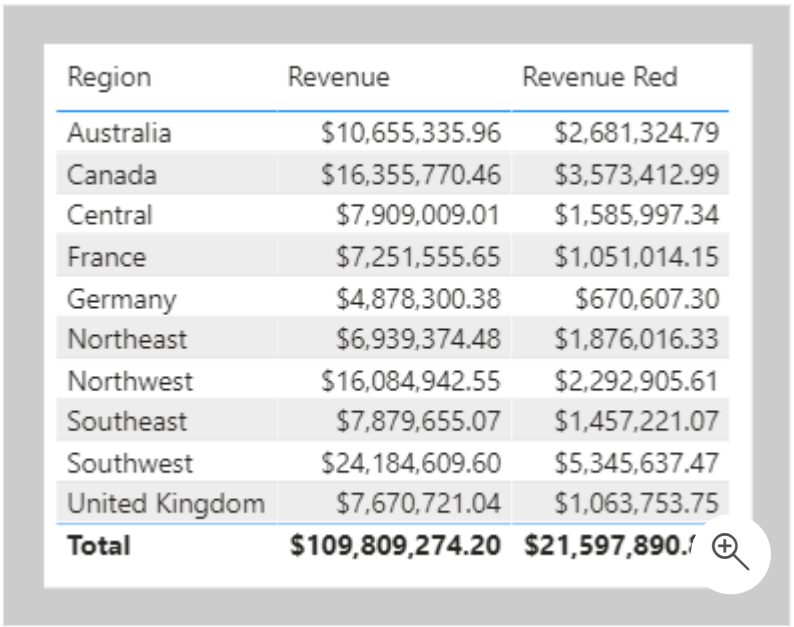
Two possible standard outcomes occur when you add filter expressions to the `CALCULATE` function:

- If the columns (or tables) aren't in filter context, then new filters will be added to the filter context to evaluate the `CALCULATE` expression.
- If the columns (or tables) are already in filter context, the existing filters will be overwritten by the new filters to evaluate the `CALCULATE` expression.

The following examples show how adding filter expressions to the `CALCULATE` function works.

> **Note**
>
> In each of the examples, no filters are applied to the table visual.

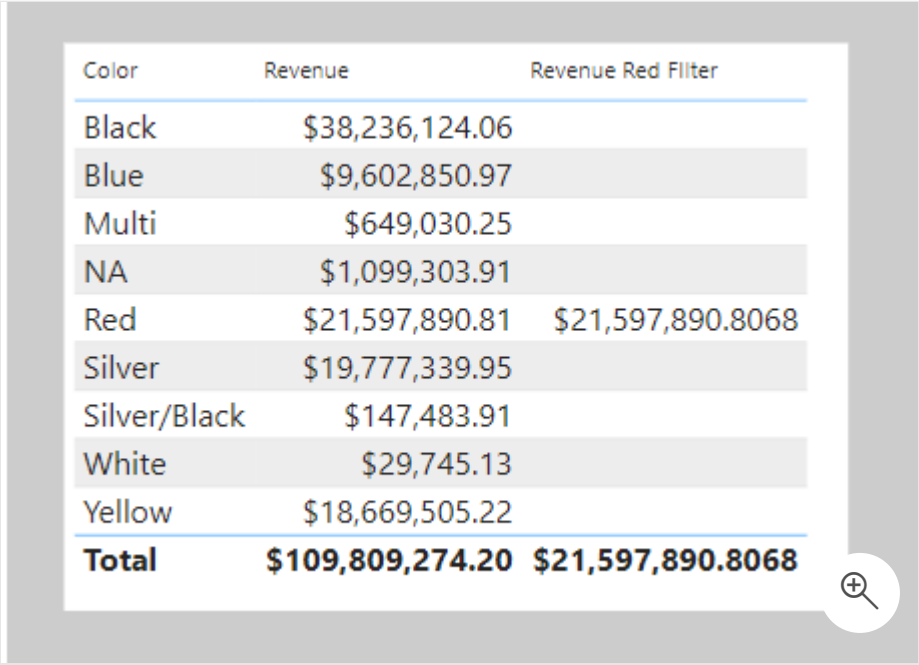
As in the previous activity, the **Revenue Red** measure was added to a table visual that groups by region and displays revenue.



Region	Revenue	Revenue Red
Australia	$10,655,335.96	$2,681,324.79
Canada	$16,355,770.46	$3,573,412.99
Central	$7,909,009.01	$1,585,997.34
France	$7,251,555.65	$1,051,014.15
Germany	$4,878,300.38	$670,607.30
Northeast	$6,939,374.48	$1,876,016.33
Northwest	$16,084,942.55	$2,292,905.61
Southeast	$7,879,655.07	$1,457,221.07
Southwest	$24,184,609.60	$5,345,637.47
United Kingdom	$7,670,721.04	$1,063,753.75
Total	**$109,809,274.20**	**$21,597,890.**

Because no filter is applied on the **Color** column in the **Product** table, the evaluation of the measure adds a new filter to filter context. In the first row, the value of $2,681,324.79 is for red products that were sold in the Australian region.

Switching the first column of the table visual from **Region** to **Color** will produce a different result because the **Color** column in the **Product** table is now in filter context.



Color	Revenue	Revenue Red Filter
Black	$38,236,124.06	
Blue	$9,602,850.97	
Multi	$649,030.25	
NA	$1,099,303.91	
Red	$21,597,890.81	$21,597,890.8068
Silver	$19,777,339.95	
Silver/Black	$147,483.91	
White	$29,745.13	
Yellow	$18,669,505.22	
Total	**$109,809,274.20**	**$21,597,890.8068**

The **Revenue Red** measure formula evaluates the **Revenue** measure by adding a filter on the **Color** column (to red) in the **Product** table. Consequently, in this visual that groups by color, the measure formula overwrites the filter context with a new filter.

This result might or might not be what you want. The next unit introduces the KEEPFILTERS DAX function, which is a filter modification function that you can use to preserve filters rather than overwrite them.

Next unit: Use filter modifier functions

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Use filter modifier functions

9 minutes

When using the `CALCULATE` function, you can pass in filter modification functions, which allow you to accomplish more than adding filters alone.

Remove filters

Use the REMOVEFILTERS DAX function as a `CALCULATE` filter expression to remove filters from filter context. It can remove filters from one or more columns or from all columns of a single table.

> **Note**
>
> The `REMOVEFILTERS` function is relatively new. In previous versions of DAX, you removed filters by using the **ALL** DAX function or variants including the **ALLEXCEPT** and the **ALLNOBLANKROW** DAX functions. These functions behave as both filter modifiers and as functions that return table objects of distinct values. These functions are mentioned now because you're likely to find documentation and formula examples that remove filters by using them.

In the following example, you will add a new measure to the **Sales** table that evaluates the **Revenue** measure but does so by removing filters from the **Sales Territory** table. Format the measure as currency with two decimal places.

DAX

```
Revenue Total Region = CALCULATE([Revenue], REMOVEFILTERS('Sales Territory'))
```

Now, add the **Revenue Total Region** measure to the matrix visual that is found on **Page 2** of the report. The matrix visual will group by three columns from the Sales Territory table on the rows: **Group**, **Country**, and **Region**.

Reseller Revenue

Group	Country	Region	Revenue	Revenue Total Region
Corporate HQ	Corporate HQ	Corporate HQ		$80,450,596.98
		Total		**$80,450,596.98**
	Total			**$80,450,596.98**
Europe	France	France	$4,607,537.94	$80,450,596.98
		Total	**$4,607,537.94**	**$80,450,596.98**
	Germany	Germany	$1,983,988.04	$80,450,596.98
		Total	**$1,983,988.04**	**$80,450,596.98**
	United Kingdom	United Kingdom	$4,279,008.83	$80,450,596.98
		Total	**$4,279,008.83**	**$80,450,596.98**
	Total		**$10,870,534.80**	**$80,450,596.98**
North America	Canada	Canada	$14,377,925.60	$80,450,596.98
		Total	**$14,377,925.60**	**$80,450,596.98**
	United States	Central	$7,906,008.18	$80,450,596.98
		Northeast	$6,932,842.01	$80,450,596.98
		Northwest	$12,435,076.00	$80,450,596.98
		Southeast	$7,867,416.23	$80,450,596.98
		Southwest	$18,466,458.79	$80,450,596.98
		Total	**$53,607,801.21**	**$80,450,596.98**
	Total		**$67,985,726.81**	**$80,450,596.98**
Pacific	Australia	Australia	$1,594,335.38	$80,450,596.98
		Total	**$1,594,335.38**	**$80,450,596.98**
	Total		**$1,594,335.38**	**$80,450,596.98**
Total			**$80,450,596.98**	**$80,450,596.98**

Notice that each **Revenue Total Region** value is the same. It's the value of total revenue.

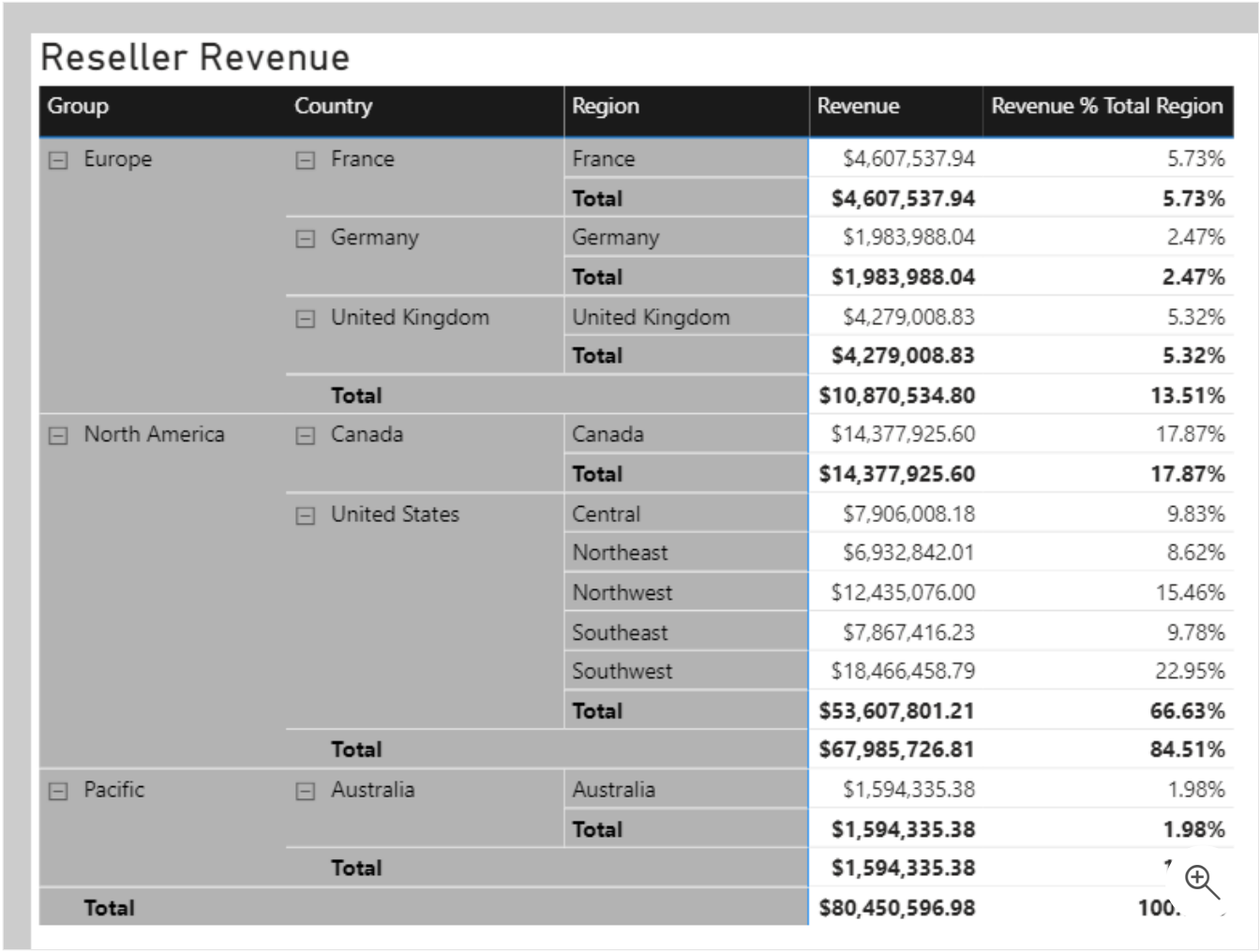
While this result on its own isn't useful, when it's used as a denominator in a ratio, it calculates a percent of grand total. Therefore, you will now overwrite the **Revenue Total Region** measure definition with the following definition. (This new definition changes the measure name and declares two variables. Be sure to format the measure as a percentage with two decimal places.)

DAX

```
Revenue % Total Region =
VAR CurrentRegionRevenue = [Revenue]
VAR TotalRegionRevenue =
    CALCULATE(
        [Revenue],
        REMOVEFILTERS('Sales Territory')
    )
RETURN
```

```
    DIVIDE(
        CurrentRegionRevenue,
        TotalRegionRevenue
    )
```

Verify that the matrix visual now displays the **Revenue % Total Region** values.



Reseller Revenue

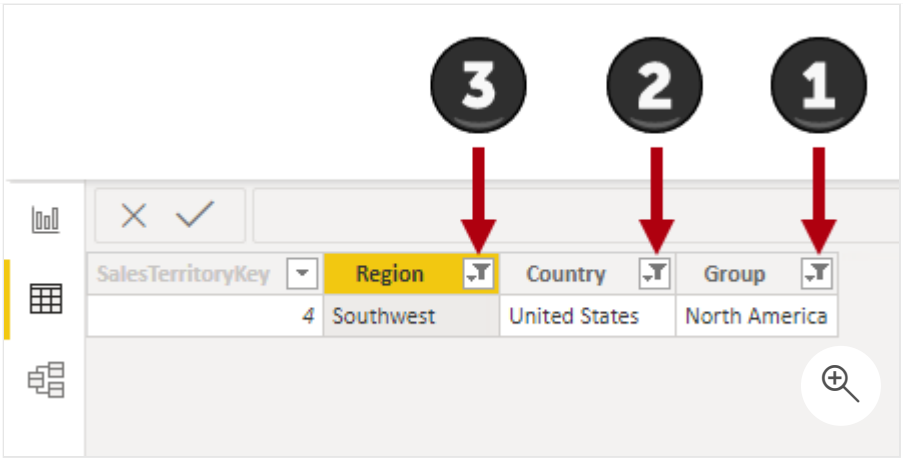
Group	Country	Region	Revenue	Revenue % Total Region
Europe	France	France	$4,607,537.94	5.73%
		Total	**$4,607,537.94**	**5.73%**
	Germany	Germany	$1,983,988.04	2.47%
		Total	**$1,983,988.04**	**2.47%**
	United Kingdom	United Kingdom	$4,279,008.83	5.32%
		Total	**$4,279,008.83**	**5.32%**
	Total		**$10,870,534.80**	**13.51%**
North America	Canada	Canada	$14,377,925.60	17.87%
		Total	**$14,377,925.60**	**17.87%**
	United States	Central	$7,906,008.18	9.83%
		Northeast	$6,932,842.01	8.62%
		Northwest	$12,435,076.00	15.46%
		Southeast	$7,867,416.23	9.78%
		Southwest	$18,466,458.79	22.95%
		Total	**$53,607,801.21**	**66.63%**
	Total		**$67,985,726.81**	**84.51%**
Pacific	Australia	Australia	$1,594,335.38	1.98%
		Total	**$1,594,335.38**	**1.98%**
	Total		**$1,594,335.38**	**1**
Total			**$80,450,596.98**	**100.**

You'll now create another measure, but this time, you will calculate the ratio of revenue for a region divided by its country's revenue.

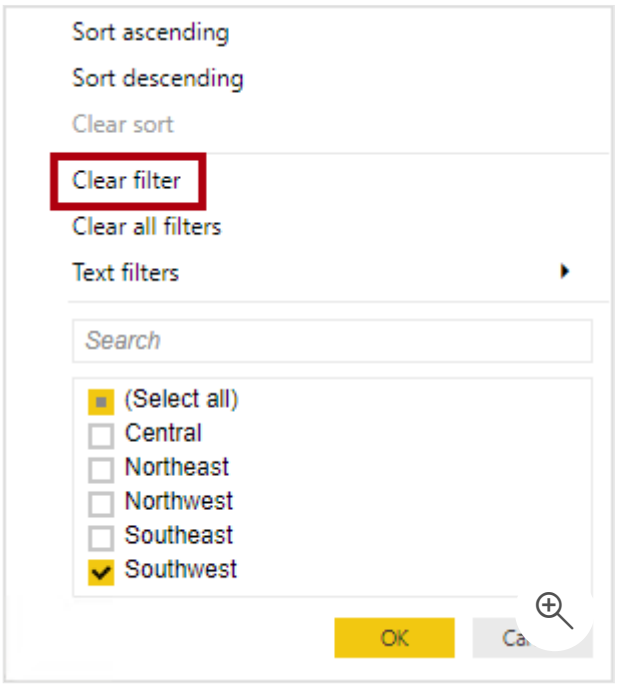
Before you complete this task, notice that the **Revenue % Total Region** value for the Southwest region is 22.95 percent. Investigate the filter context for this cell. Switch to data view and then, in the **Fields** pane, select the **Sales Territory** table.

Apply the following column filters:

- **Group** - North America
- **Country** - United States
- **Region** - Southwest



Notice that the filters reduce the table to only one row. Now, while thinking about your new objective to create a ratio of the region revenue over its country's revenue, clear the filter from the **Region** column.



Notice that five rows now exist, each row belonging to the country United States. Accordingly, when you clear the **Region** column filters, while preserving filters on the **Country** and **Group** columns, you will have a new filter context that's for the region's country.

In the following measure definition, notice how you can clear or remove a filter from a column. In DAX logic, it's a small and subtle change that's made to the **Revenue % Total Region** measure formula: The `REMOVEFILTERS` function now removes filters from the **Region** column instead of all columns of the **Sales Territory** table.

DAX

```
Revenue % Total Country =
VAR CurrentRegionRevenue = [Revenue]
VAR TotalCountryRevenue =
    CALCULATE(
```

```
        [Revenue],
        REMOVEFILTERS('Sales Territory'[Region])
    )
RETURN
    DIVIDE(
        CurrentRegionRevenue,
        TotalCountryRevenue
    )
```

Add the **Revenue % Total Country** measure to the **Sales** table and then format it as a percentage with two decimal places. Add the new measure to the matrix visual.

Reseller Revenue

Group	Country	Region	Revenue	Revenue % Total Region	Revenue % Total Country
Europe	France	France	$4,607,537.94	5.73%	100.00%
		Total	**$4,607,537.94**	**5.73%**	**100.00%**
	Germany	Germany	$1,983,988.04	2.47%	100.00%
		Total	**$1,983,988.04**	**2.47%**	**100.00%**
	United Kingdom	United Kingdom	$4,279,008.83	5.32%	100.00%
		Total	**$4,279,008.83**	**5.32%**	**100.00%**
	Total		**$10,870,534.80**	**13.51%**	**100.00%**
North America	Canada	Canada	$14,377,925.60	17.87%	100.00%
		Total	**$14,377,925.60**	**17.87%**	**100.00%**
	United States	Central	$7,906,008.18	9.83%	14.75%
		Northeast	$6,932,842.01	8.62%	12.93%
		Northwest	$12,435,076.00	15.46%	23.20%
		Southeast	$7,867,416.23	9.78%	14.68%
		Southwest	$18,466,458.79	22.95%	34.45%
		Total	**$53,607,801.21**	**66.63%**	**100.00%**
	Total		**$67,985,726.81**	**84.51%**	**100.00%**
Pacific	Australia	Australia	$1,594,335.38	1.98%	100.00%
		Total	**$1,594,335.38**	**1.98%**	**100.00%**
	Total		**$1,594,335.38**	**1.98%**	**100**
Total			**$80,450,596.98**	**100.00%**	**100.**

Notice that all values, except those values for United States regions, are 100 percent. The reason is because, at the Adventure Works company, the United States has regions, while all other countries do not.

> **Note**
>
> Tabular models don't support ragged hierarchies, which are hierarchies with variable depths. Therefore, it's a common design approach to repeat parent (or other ancestor) values at lower levels of the hierarchy. For example, Australia doesn't have a region, so the country/region value is repeated as the region name. It's always better to store a meaningful value instead of BLANK.

The next example is last measure that you will create. Add the **Revenue % Total Group** measure, and then format it as a percentage with two decimal places. Then, add the new measure to the matrix visual.

DAX

```
Revenue % Total Group =
VAR CurrentRegionRevenue = [Revenue]
VAR TotalGroupRevenue =
    CALCULATE(
        [Revenue],
        REMOVEFILTERS(
            'Sales Territory'[Region],
            'Sales Territory'[Country]
        )
    )
RETURN
    DIVIDE(
        CurrentRegionRevenue,
        TotalGroupRevenue
    )
```

Reseller Revenue

Group	Country	Region	Revenue	Revenue % Total Region	Revenue % Total Country	Revenue % Total Group
Europe	France	France	$4,607,537.94	5.73%	100.00%	42.39%
		Total	**$4,607,537.94**	**5.73%**	**100.00%**	**42.39%**
	Germany	Germany	$1,983,988.04	2.47%	100.00%	18.25%
		Total	**$1,983,988.04**	**2.47%**	**100.00%**	**18.25%**
	United Kingdom	United Kingdom	$4,279,008.83	5.32%	100.00%	39.36%
		Total	**$4,279,008.83**	**5.32%**	**100.00%**	**39.36%**
	Total		**$10,870,534.80**	**13.51%**	**100.00%**	**100.00%**
North America	Canada	Canada	$14,377,925.60	17.87%	100.00%	21.15%
		Total	**$14,377,925.60**	**17.87%**	**100.00%**	**21.15%**
	United States	Central	$7,906,008.18	9.83%	14.75%	11.63%
		Northeast	$6,932,842.01	8.62%	12.93%	10.20%
		Northwest	$12,435,076.00	15.46%	23.20%	18.29%
		Southeast	$7,867,416.23	9.78%	14.68%	11.57%
		Southwest	$18,466,458.79	22.95%	34.45%	27.16%
		Total	**$53,607,801.21**	**66.63%**	**100.00%**	**78.85%**
	Total		**$67,985,726.81**	**84.51%**	**100.00%**	**100.00%**
Pacific	Australia	Australia	$1,594,335.38	1.98%	100.00%	100.00%
		Total	**$1,594,335.38**	**1.98%**	**100.00%**	**100.**
	Total		**$1,594,335.38**	**1.98%**	**100.00%**	**100**
Total			**$80,450,596.98**	**100.00%**	**100.00%**	**100.0**

When you remove filters from the **Region** and **Country** columns in the **Sales Territory** table, the measure will calculate the region revenue as a ratio of its group's revenue.

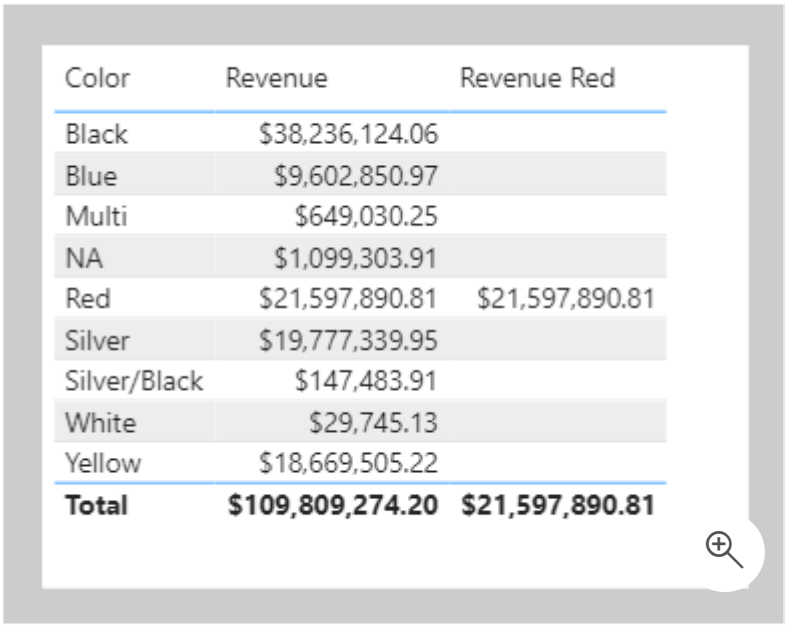
Preserve filters

You can use the KEEPFILTERS DAX function as a filter expression in the `CALCULATE` function to preserve filters.

To observe how to accomplish this task, switch to **Page 1** of the report. Then, modify the **Revenue Red** measure definition to use the `KEEPFILTERS` function.

DAX

```
Revenue Red =
CALCULATE(
    [Revenue],
    KEEPFILTERS('Product'[Color] = "Red")
)
```



Color	Revenue	Revenue Red
Black	$38,236,124.06	
Blue	$9,602,850.97	
Multi	$649,030.25	
NA	$1,099,303.91	
Red	$21,597,890.81	$21,597,890.81
Silver	$19,777,339.95	
Silver/Black	$147,483.91	
White	$29,745.13	
Yellow	$18,669,505.22	
Total	**$109,809,274.20**	**$21,597,890.81**

In the table visual, notice that only one **Revenue Red** value exists. The reason is because the Boolean filter expression preserves existing filters on the **Color** column in the **Product** table. The reason why colors other than red are BLANK is because the filter contexts and the filter expressions are combined for these two filters. The color black and color red are intersected, and because both can't be `TRUE` at the same time, the expression is filtered by no product rows. It's only possible that both red filters can be `TRUE` at the same time, which explains why the one **Revenue Red** value is shown.

Use inactive relationships

An inactive model relationship can only propagate filters when the USERELATIONSHIP DAX function is passed as a filter expression to the `CALCULATE` function. When you use this function to engage an inactive relationship, the active relationship will automatically become inactive.

Review an example of a measure definition that uses an inactive relationship to calculate the **Revenue** measure by shipped dates:

DAX

```
Revenue Shipped =
CALCULATE (
    [Revenue],
    USERELATIONSHIP('Date'[DateKey], Sales[ShipDateKey])
)
```

Modify relationship behavior

You can modify the model relationship behavior when an expression is evaluated by passing the CROSSFILTER DAX function as a filter expression to the `CALCULATE` function. It's an advanced capability.

The `CROSSFILTER` function can modify filter directions (from both to single or from single to both) and even disable a relationship.

Next unit: Examine filter context

Continue >

---

100 XP

Examine filter context

4 minutes

The VALUES DAX function lets your formulas determine what values are in filter context.

The `VALUES` function syntax is as follows:

```DAX
VALUES(<TableNameOrColumnName>)
```

The function requires passing in a table reference or a column reference. When you pass in a table reference, it returns a table object with the same columns that contain rows for what's in filter context. When you pass in a column reference, it returns a single-column table of unique values that are in filter context.

The function always returns a table object and it's possible for a table to contain multiple rows. Therefore, to test whether a specific value is in filter context, your formula must first test that the `VALUES` function returns a single row. Two functions can help you accomplish this task: the HASONEVALUE and the SELECTEDVALUE DAX functions.

The `HASONEVALUE` function returns `TRUE` when a given column reference has been filtered down to a single value.

The `SELECTEDVALUE` function simplifies the task of determining what a single value could be. When the function is passed a column reference, it'll return a single value, or when more than one value is in filter context, it'll return BLANK (or an alternate value that you pass to the function).

In the following example, you will use the `HASONEVALUE` function. Add the following measure, which calculates sales commission, to the **Sales** table. Note that, at Adventure Works, the commission rate is 10 percent of revenue for all countries/regions except the United States. In the United States, salespeople earn 15 percent commission. Format the measure as currency with two decimal places, and then add it to the table that is found on **Page 3** of the report.

```DAX
Sales Commission =
[Revenue]
    * IF(

        HASONEVALUE('Sales Territory'[Country])
```

```
        HASONEVALUE( 'Sales Territory'[Country]),
        IF(
            VALUES('Sales Territory'[Country]) = "United States",
            0.15,
            0.1
        )
    )
```

Region	Revenue	Sales Commission
Australia	$10,655,335.96	$1,065,533.60
Canada	$16,355,770.46	$1,635,577.05
Central	$7,909,009.01	$1,186,351.35
France	$7,251,555.65	$725,155.56
Germany	$4,878,300.38	$487,830.04
Northeast	$6,939,374.48	$1,040,906.17
Northwest	$16,084,942.55	$2,412,741.38
Southeast	$7,879,655.07	$1,181,948.26
Southwest	$24,184,609.60	$3,627,691.44
United Kingdom	$7,670,721.04	$767,072.10
Total	**$109,809,274.20**	

Notice that the total **Sales Commission** result is BLANK. The reason is because multiple values are in filter context for the **Country** column in the **Sales Territory** table. In this case, the `HASONEVALUE` function returns `FALSE`, which results in the **Revenue** measure being multiplied by BLANK (a value multiplied by BLANK is BLANK). To produce a total, you will need to use an iterator function, which is explained later in this module.

Three other functions that you can use to test filter state are:

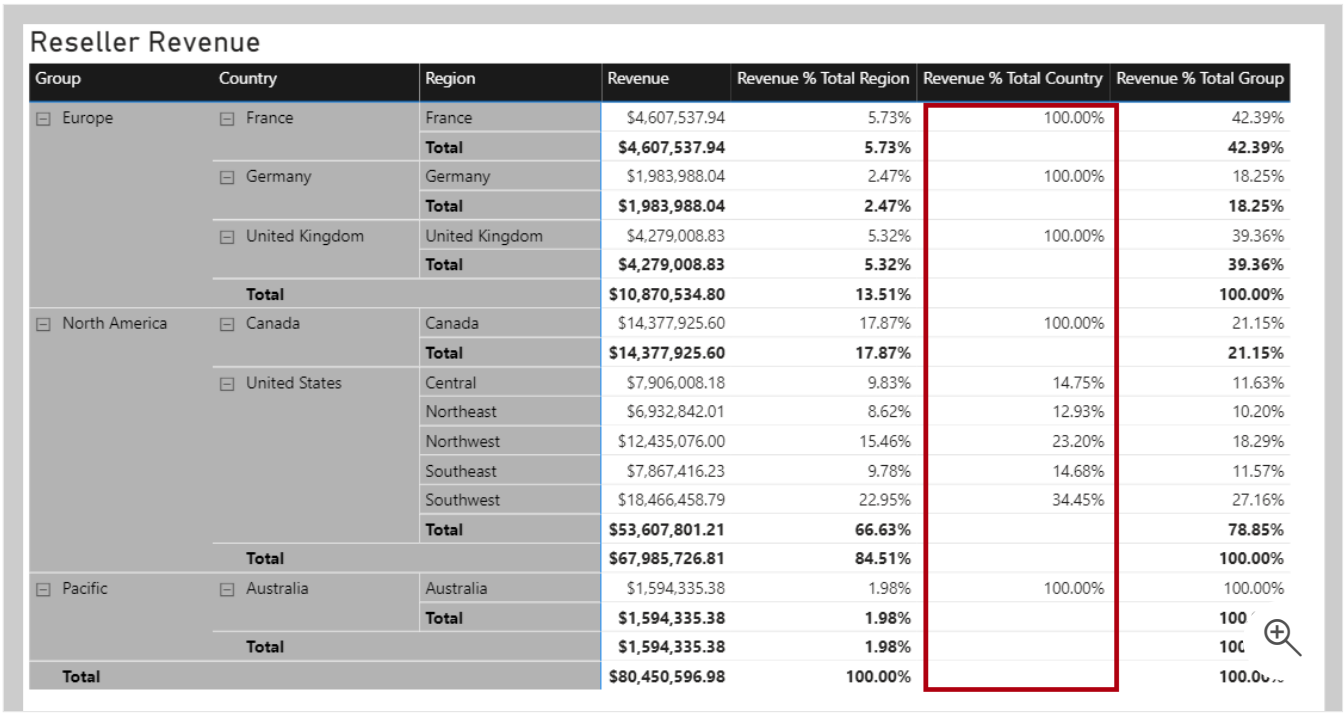
- ISFILTERED - Returns `TRUE` when a passed-in column reference is *directly* filtered.
- ISCROSSFILTERED - Returns `TRUE` when a passed-in column reference is *indirectly* filtered. A column is cross-filtered when a filter that is applied to another column in the same table, or in a related table, affects the reference column by filtering it.
- ISINSCOPE - Returns `TRUE` when a passed-in column reference is the level in a hierarchy of levels.

Return to **Page 2** of the report, and then modify the **Revenue % Total Country** measure definition to test that the **Region** column in the **Sales Territory** table is in scope. If it's not in scope, the measure result should be BLANK.

DAX

```
Revenue % Total Country =
VAR CurrentRegionRevenue = [Revenue]
VAR TotalCountryRevenue =
```

```
VAR TotalCountryRevenue =
    CALCULATE(
        [Revenue],
        REMOVEFILTERS('Sales Territory'[Region])
    )
RETURN
    IF(
        ISINSCOPE('Sales Territory'[Region]),
        DIVIDE(
            CurrentRegionRevenue,
            TotalCountryRevenue
        )
    )
```



Reseller Revenue

Group	Country	Region	Revenue	Revenue % Total Region	Revenue % Total Country	Revenue % Total Group
Europe	France	France	$4,607,537.94	5.73%	100.00%	42.39%
		Total	**$4,607,537.94**	**5.73%**		**42.39%**
	Germany	Germany	$1,983,988.04	2.47%	100.00%	18.25%
		Total	**$1,983,988.04**	**2.47%**		**18.25%**
	United Kingdom	United Kingdom	$4,279,008.83	5.32%	100.00%	39.36%
		Total	**$4,279,008.83**	**5.32%**		**39.36%**
	Total		**$10,870,534.80**	**13.51%**		**100.00%**
North America	Canada	Canada	$14,377,925.60	17.87%	100.00%	21.15%
		Total	**$14,377,925.60**	**17.87%**		**21.15%**
	United States	Central	$7,906,008.18	9.83%	14.75%	11.63%
		Northeast	$6,932,842.01	8.62%	12.93%	10.20%
		Northwest	$12,435,076.00	15.46%	23.20%	18.29%
		Southeast	$7,867,416.23	9.78%	14.68%	11.57%
		Southwest	$18,466,458.79	22.95%	34.45%	27.16%
		Total	**$53,607,801.21**	**66.63%**		**78.85%**
	Total		**$67,985,726.81**	**84.51%**		**100.00%**
Pacific	Australia	Australia	$1,594,335.38	1.98%	100.00%	100.00%
		Total	**$1,594,335.38**	**1.98%**		**100**
	Total		**$1,594,335.38**	**1.98%**		**10**
Total			**$80,450,596.98**	**100.00%**		**100.0**

In the matrix visual, notice that **Revenue % Total Country** values are now only displayed when a region is in scope.

Next unit: Perform context transition

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Perform context transition

5 minutes

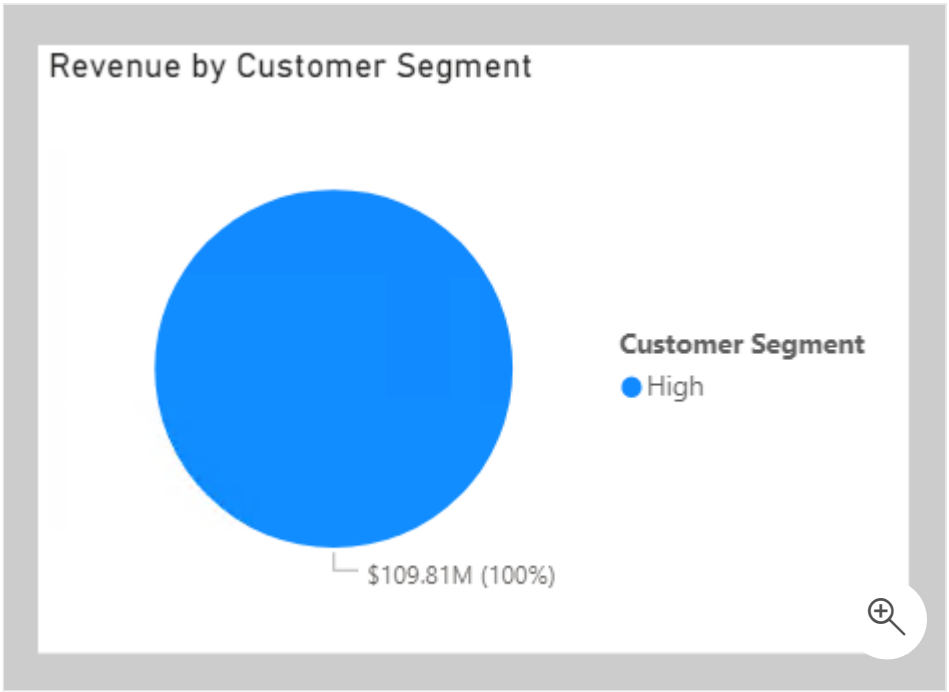
What happens when a measure or measure expression is evaluated within row context? This scenario can happen in a calculated column formula or when an expression in an iterator function is evaluated.

In the following example, you will add a calculated column to the **Customer** table to classify customers into a loyalty class. The scenario is simple: When the revenue that is produced by the customer is less than $2500, the customer is classified as **Low**; otherwise they're classified as **High**.

DAX

```
Customer Segment =
VAR CustomerRevenue = SUM(Sales[Sales Amount])
RETURN
    IF(CustomerRevenue < 2500, "Low", "High")
```

On **Page 4** of the report, add the **Customer Segment** column as the legend of the pie chart.



Notice that only one **Customer Segment** value exists. The reason is because the calculated column formula produces an incorrect result: Each customer is assigned the value of **High** because the expression `SUM(Sales[Sales Amount])` isn't evaluated in a filter context.

Consequently, each customer is assessed on the sum of *every* **Sales Amount** column value in the **Sales** table.

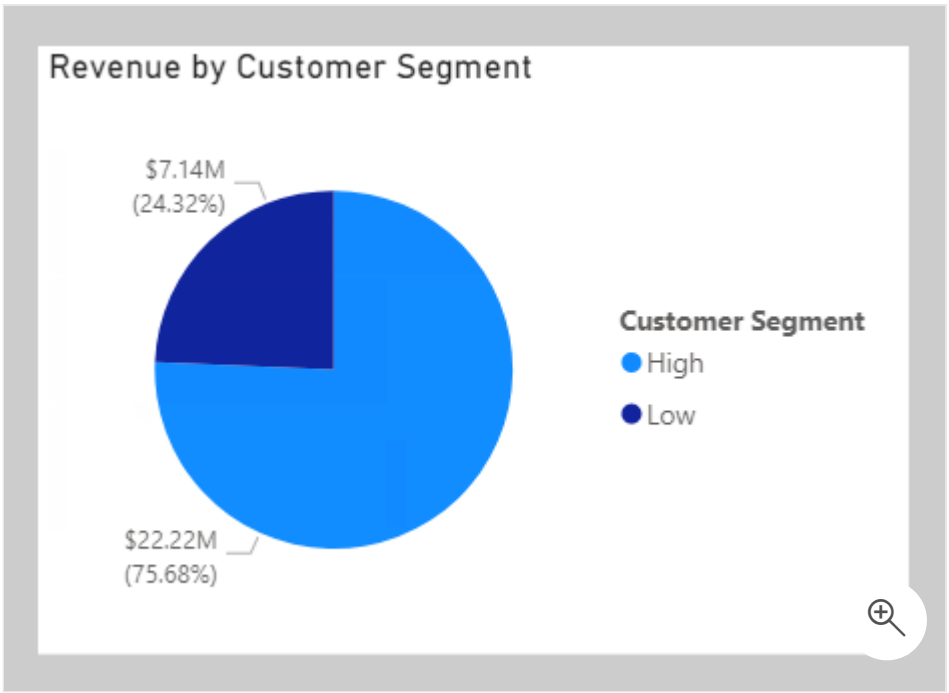
To force the evaluation of the `SUM(Sales[Sales Amount])` expression *for each customer*, a context transition must take place that applies the row context column values to filter context. You can accomplish this transition by using the `CALCULATE` function without passing in filter expressions.

Modify the calculated column definition so that it produces the correct result.

DAX

```
Customer Segment =
VAR CustomerRevenue = CALCULATE(SUM(Sales[Sales Amount]))
RETURN
    IF(CustomerRevenue < 2500, "Low", "High")
```

In the pie chart visual add the new calculated column to the Legend well, verify that two pie segments now display.



In this case, the `CALCULATE` function applies row context values as filters, known as *context transition*. To be accurate, the process doesn't quite work that way when a unique column is on the table. When a unique column is on the table, you only need to apply a filter on that column to make the transition happen. In this case, Power BI applies a filter on the **CustomerKey** column for the value in row context.

If you reference measures in an expression that's evaluated in row context, context transition is automatic. Thus, you don't need to pass measure references to the `CALCULATE` function.

Modify the calculated column definition, which references the **Revenue** measure, and notice that it continues to produce the correct result.

DAX

```
Customer Segment =
VAR CustomerRevenue = [Revenue]
RETURN
    IF(CustomerRevenue < 2500, "Low", "High")
```

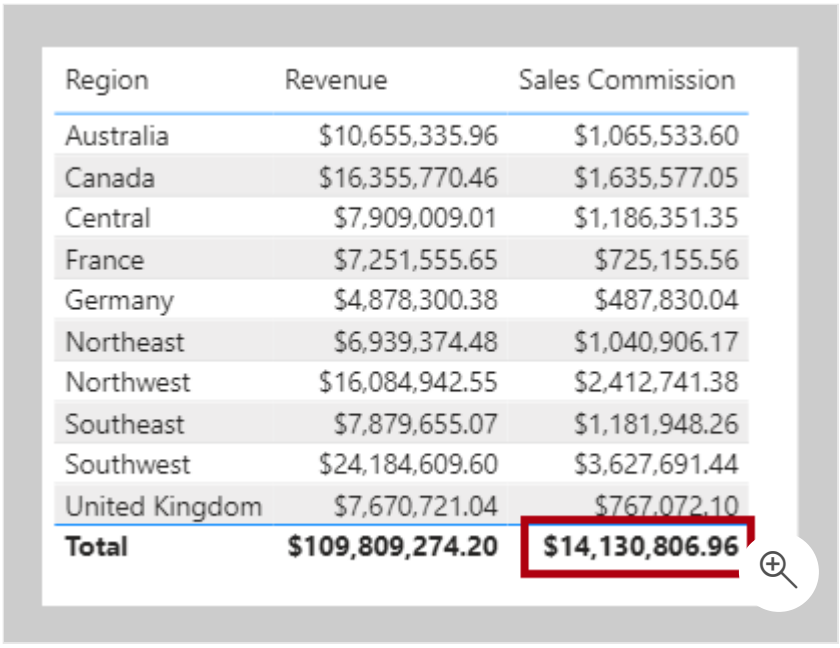
Now, you can complete the **Sales Commission** measure formula. To produce a total, you need to use an iterator function to iterate over all regions in filter context. The iterator function expression must use the `CALCULATE` function to transition the row context to the filter context. Notice that it no longer needs to test whether a single **Country** column value in the **Sales Territory** table is in filter context because it's known to be filtering by a single country (because it's iterating over the regions in filter context and a region belongs to only one country).

Switch to **Page 3** of the report, and then modify the **Sales Commission** measure definition to use the `SUMX` iterator function:

DAX

```
Sales Commission =
SUMX(
    VALUES('Sales Territory'[Region]),
    CALCULATE(
        [Revenue]
        * IF(
            VALUES('Sales Territory'[Country]) = "United States",
            0.15,
            0.1
        )
    )
)
```

The table visual now displays a sales commission total for all regions.



Region	Revenue	Sales Commission
Australia	$10,655,335.96	$1,065,533.60
Canada	$16,355,770.46	$1,635,577.05
Central	$7,909,009.01	$1,186,351.35
France	$7,251,555.65	$725,155.56
Germany	$4,878,300.38	$487,830.04
Northeast	$6,939,374.48	$1,040,906.17
Northwest	$16,084,942.55	$2,412,741.38
Southeast	$7,879,655.07	$1,181,948.26
Southwest	$24,184,609.60	$3,627,691.44
United Kingdom	$7,670,721.04	$767,072.10
Total	**$109,809,274.20**	**$14,130,806.96**

Next unit: Check your knowledge

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

1. Which type of model object is evaluated within a filter context? *

- ○ Calculated column
  - ✗ Calculated columns are evaluated within row context. It's possible, however, to transition row context to filter context when a measure expression is evaluated.
- ○ Calculated table
- ○ Measure
  - ✓ Measures (or measure expressions) are always evaluated in filter context.
- ○ Security role rule

2. Which one of the following DAX functions allows you to use an inactive relationship when evaluating a measure expression? *

- ○ USERELATIONSHIP
  - ✓ The USERELATIONSHIP function is a filter modifier function that can be passed in to the CALCULATE function. Its purpose is to engage an inactive relationship.
- ○ CROSSFILTER
  - ✗ The CROSSFILTER function isn't used to engage an inactive relationship. It's used to change relationship filter directions or disable a relationship.
- ○ REMOVEFILTERS
- ○ ISCROSSFILTERED

3. Which one of the following statements about the CALCULATE function is true? *

- ○ You must pass in at least one filter argument.
  - ✗ The CALCULATE function, when used to transition row context to filter context, does not require filter arguments.
- ○ It modifies filter context to evaluate a given expression.
  - ✓ The CALCULATE function modifies filter context by adding or removing filters or by modifying standard filter behavior.
- ○ It modifies row context to evaluate a given expression.

It can't be used by a calculated column formula.

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

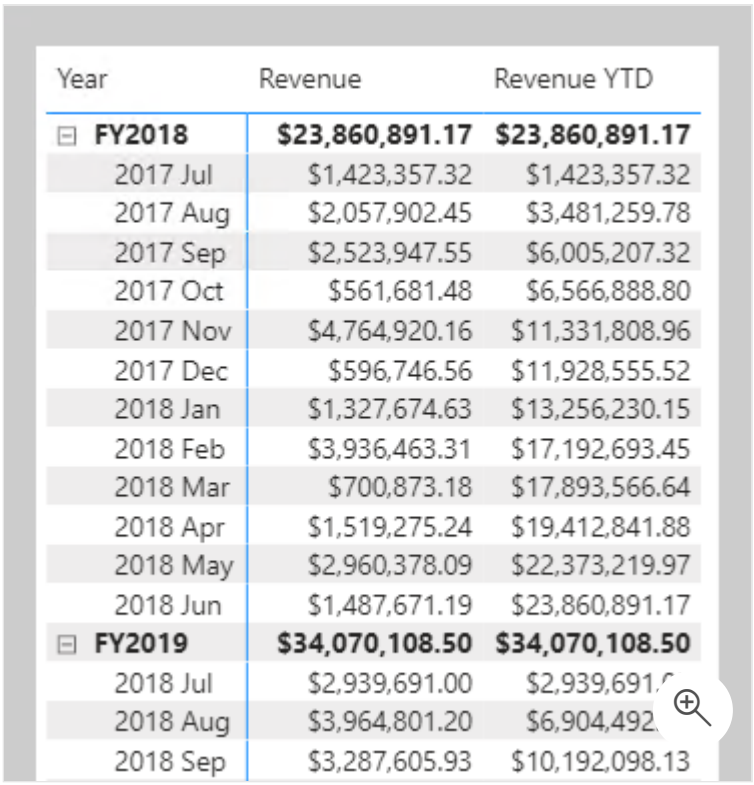
Introduction

1 minute

Time intelligence relates to calculations over time. Specifically, it relates to calculations over dates, months, quarters, or years, and possibly time. Rarely would you need to calculate over time in the sense of hours, minutes, or seconds.

In Data Analysis Expressions (DAX) calculations, time intelligence means *modifying the filter context for date filters*.

For example, at the Adventure Works company, their financial year begins on July 1 and ends on June 30 of the following year. They produce a table visual that displays monthly revenue and year-to-date (YTD) revenue.



Year	Revenue	Revenue YTD
FY2018	**$23,860,891.17**	**$23,860,891.17**
2017 Jul	$1,423,357.32	$1,423,357.32
2017 Aug	$2,057,902.45	$3,481,259.78
2017 Sep	$2,523,947.55	$6,005,207.32
2017 Oct	$561,681.48	$6,566,888.80
2017 Nov	$4,764,920.16	$11,331,808.96
2017 Dec	$596,746.56	$11,928,555.52
2018 Jan	$1,327,674.63	$13,256,230.15
2018 Feb	$3,936,463.31	$17,192,693.45
2018 Mar	$700,873.18	$17,893,566.64
2018 Apr	$1,519,275.24	$19,412,841.88
2018 May	$2,960,378.09	$22,373,219.97
2018 Jun	$1,487,671.19	$23,860,891.17
FY2019	**$34,070,108.50**	**$34,070,108.50**
2018 Jul	$2,939,691.00	$2,939,691.(
2018 Aug	$3,964,801.20	$6,904,492.
2018 Sep	$3,287,605.93	$10,192,098.13

The filter context for **2017 August** contains each of the 31 dates of August, which are stored in the **Date** table. However, the calculated year-to-date revenue for **2017 August** applies a different filter context. It's the first date of the year through to the last date in filter context. In this example, that's July 1, 2017 through to August 31, 2017.

Time intelligence calculations modify date filter contexts. They can help you answer these time-related questions:

- What's the accumulation of revenue for the year, quarter, or month?

- What revenue was produced for the same period last year?
- What growth in revenue has been achieved over the same period last year?
- How many new customers made their first order in each month?
- What's the inventory stock on-hand value for the company's products?

This module describes how to create time intelligence measures to answer these questions.

Next unit: Use DAX time intelligence functions

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

Use DAX time intelligence functions

5 minutes

DAX includes several time intelligence functions to simplify the task of modifying date filter context. You could write many of these intelligence formulas by using a `CALCULATE` function that modifies date filters, but that would create more work.

> **Note**
>
> Many DAX time intelligence functions are concerned with standard date periods, specifically years, quarters, and months. If you have irregular time periods (for example, financial months that begin mid-way through the calendar month), or you need to work with weeks or time periods (hours, minutes, and so on), the DAX time intelligence functions won't be helpful. Instead, you'll need to use the `CALCULATE` function and pass in hand-crafted date or time filters.

Date table requirement

To work with time intelligence DAX functions, you need to meet the prerequisite model requirement of having at least one *date table* in your model. A date table is a table that meets the following requirements:

- It must have a column of data type Date (or date/time), known as the *date column*.
- The date column must contain unique values.
- The date column must not contain BLANKs.
- The date column must not have any missing dates.
- The date column must span full years. A year isn't necessarily a calendar year (January-December).
- The date table must be indicated as a date table.

For more information, see Create date tables in Power BI Desktop.

Summarizations over time

One group of DAX time intelligence functions is concerned with summarizations over time:

- DATESYTD - Returns a single-column table that contains dates for the year-to-date (YTD) in the current filter context. This group also includes the DATESMTD and DATESQTD DAX functions for month-to-date (MTD) and quarter-to-date (QTD). You can pass these functions as filters into the CALCULATE DAX function.
- TOTALYTD - Evaluates an expression for YTD in the current filter context. The equivalent QTD and MTD DAX functions of TOTALQTD and TOTALMTD are also included.
- DATESBETWEEN - Returns a table that contains a column of dates that begins with a given start date and continues until a given end date.
- DATESINPERIOD - Returns a table that contains a column of dates that begins with a given start date and continues for the specified number of intervals.

> **Note**
>
> While the `TOTALYTD` function is simple to use, you are limited to passing in one filter expression. If you need to apply multiple filter expressions, use the `CALCULATE` function and then pass the `DATESYTD` function in as one of the filter expressions.

In the following example, you will create your first time intelligence calculation that will use the TOTALYTD function. The syntax is as follows:

DAX

```
TOTALYTD(<expression>, <dates>, [, <filter>][, <year_end_date>])
```

The function requires an expression and, as is common to all time intelligence functions, a reference to the date column of a marked date table. Optionally, a single filter expression or the year-end date can be passed in (required only when the year doesn't finish on December 31).

Download and open the **Adventure Works DW 2020 M07.pbix** file. Then, add the following measure definition to the **Sales** table that calculates YTD revenue. Format the measure as currency with two decimal places.

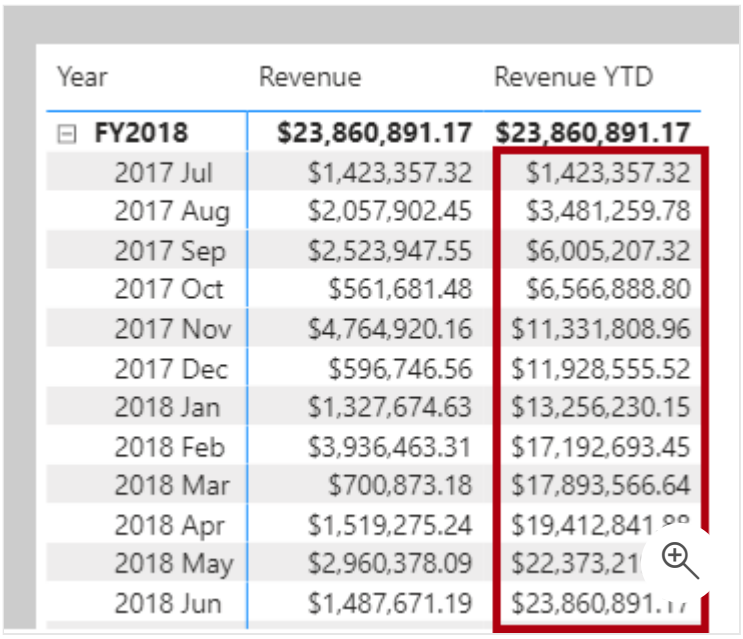
DAX

```
Revenue YTD =
TOTALYTD([Revenue], 'Date'[Date], "6-30")
```

The year-end date value of `"6-30"` represents June 30.

On **Page 1** of the report, add the **Revenue YTD** measure to the matrix visual. Notice that it produces a summarization of the revenue amounts from the beginning of the year through to

the filtered month.



Year	Revenue	Revenue YTD
FY2018	$23,860,891.17	$23,860,891.17
2017 Jul	$1,423,357.32	$1,423,357.32
2017 Aug	$2,057,902.45	$3,481,259.78
2017 Sep	$2,523,947.55	$6,005,207.32
2017 Oct	$561,681.48	$6,566,888.80
2017 Nov	$4,764,920.16	$11,331,808.96
2017 Dec	$596,746.56	$11,928,555.52
2018 Jan	$1,327,674.63	$13,256,230.15
2018 Feb	$3,936,463.31	$17,192,693.45
2018 Mar	$700,873.18	$17,893,566.64
2018 Apr	$1,519,275.24	$19,412,841.88
2018 May	$2,960,378.09	$22,373,21
2018 Jun	$1,487,671.19	$23,860,891.17

Comparisons over time

Another group of DAX time intelligence functions is concerned with shifting time periods:

- DATEADD - Returns a table that contains a column of dates, shifted either forward or backward in time by the specified number of intervals from the dates in the current filter context.
- PARALLELPERIOD - Returns a table that contains a column of dates that represents a period that is parallel to the dates in the specified dates column, in the current filter context, with the dates shifted a number of intervals either forward in time or back in time.
- SAMEPERIODLASTYEAR - Returns a table that contains a column of dates that are shifted one year back in time from the dates in the specified dates column, in the current filter context.
- Many helper DAX functions for navigating backward or forward for specific time periods, all of which returns a table of dates. These helper functions include NEXTDAY, NEXTMONTH, NEXTQUARTER, NEXTYEAR, and PREVIOUSDAY, PREVIOUSMONTH, PREVIOUSQUARTER, and PREVIOUSYEAR.

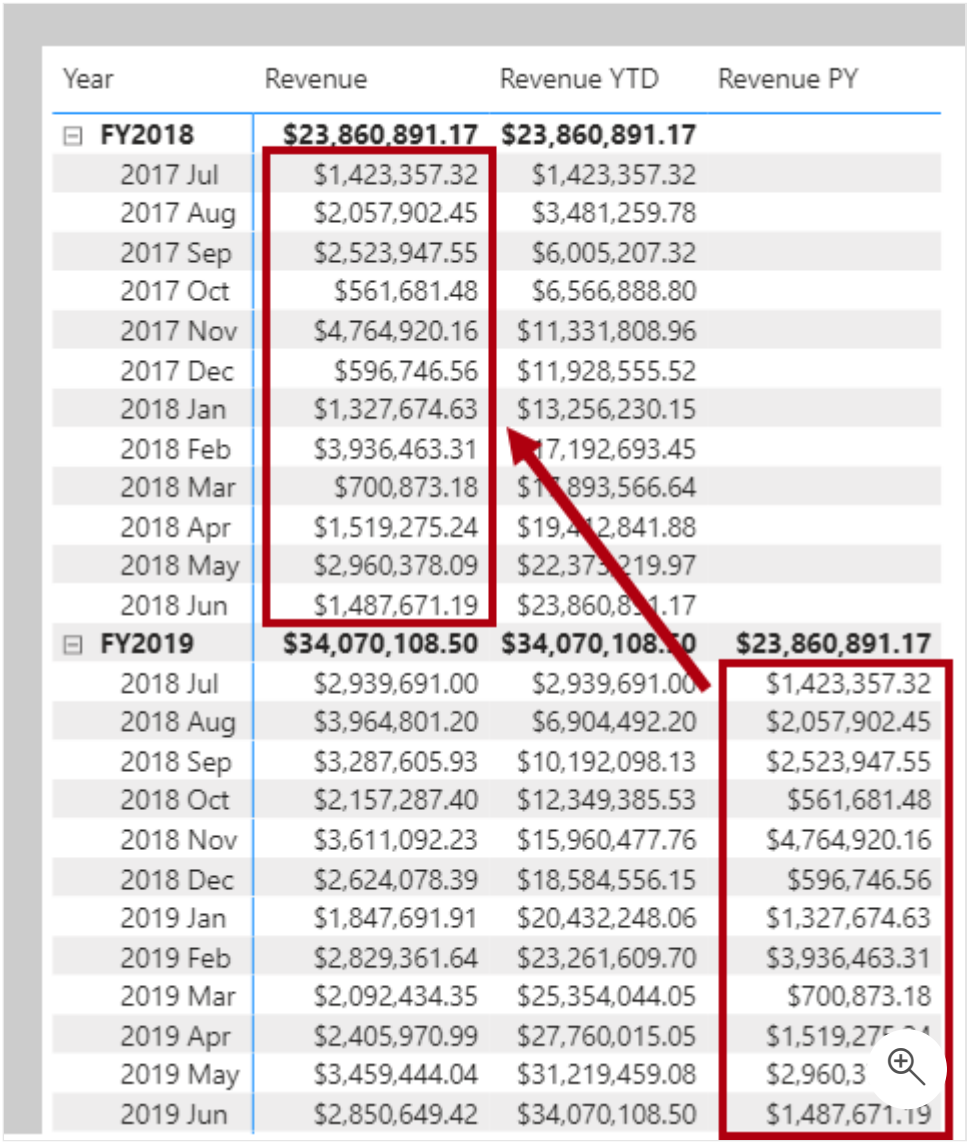
Now, you will add a measure to the **Sales** table that calculates revenue for the prior year by using the `SAMEPERIODLASTYEAR` function. Format the measure as currency with two decimal places.

DAX

```
Revenue PY =
VAR RevenuePriorYear = CALCULATE([Revenue],
```

```
SAMEPERIODLASTYEAR('Date'[Date]))
RETURN
    RevenuePriorYear
```

Add the **Revenue PY** measure to the matrix visual. Notice that it produces results that are similar to the previous year's revenue amounts.



Year	Revenue	Revenue YTD	Revenue PY
FY2018	**$23,860,891.17**	**$23,860,891.17**	
2017 Jul	$1,423,357.32	$1,423,357.32	
2017 Aug	$2,057,902.45	$3,481,259.78	
2017 Sep	$2,523,947.55	$6,005,207.32	
2017 Oct	$561,681.48	$6,566,888.80	
2017 Nov	$4,764,920.16	$11,331,808.96	
2017 Dec	$596,746.56	$11,928,555.52	
2018 Jan	$1,327,674.63	$13,256,230.15	
2018 Feb	$3,936,463.31	$17,192,693.45	
2018 Mar	$700,873.18	$17,893,566.64	
2018 Apr	$1,519,275.24	$19,412,841.88	
2018 May	$2,960,378.09	$22,373,219.97	
2018 Jun	$1,487,671.19	$23,860,891.17	
FY2019	**$34,070,108.50**	**$34,070,108.50**	**$23,860,891.17**
2018 Jul	$2,939,691.00	$2,939,691.00	$1,423,357.32
2018 Aug	$3,964,801.20	$6,904,492.20	$2,057,902.45
2018 Sep	$3,287,605.93	$10,192,098.13	$2,523,947.55
2018 Oct	$2,157,287.40	$12,349,385.53	$561,681.48
2018 Nov	$3,611,092.23	$15,960,477.76	$4,764,920.16
2018 Dec	$2,624,078.39	$18,584,556.15	$596,746.56
2019 Jan	$1,847,691.91	$20,432,248.06	$1,327,674.63
2019 Feb	$2,829,361.64	$23,261,609.70	$3,936,463.31
2019 Mar	$2,092,434.35	$25,354,044.05	$700,873.18
2019 Apr	$2,405,970.99	$27,760,015.05	$1,519,275.24
2019 May	$3,459,444.04	$31,219,459.08	$2,960,3
2019 Jun	$2,850,649.42	$34,070,108.50	$1,487,671.19

Next, you will modify the measure by renaming it to **Revenue YoY %** and then updating the `RETURN` clause to calculate the change ratio. Be sure to change the format to a percentage with two decimals places.

DAX

```
Revenue YoY % =
VAR RevenuePriorYear = CALCULATE([Revenue],
SAMEPERIODLASTYEAR('Date'[Date]))
RETURN
    DIVIDE(
        [Revenue] - RevenuePriorYear,
```

```
        RevenuePriorYear
    )
```

Notice that the **Revenue YoY %** measure produces a ratio of change factor over the previous year's monthly revenue. For example, July 2018 represents a 106.53 percent *increase* over the previous year's monthly revenue, and November 2018 represents a 24.22 percent *decrease* over the previous year's monthly revenue.

Year	Revenue	Revenue YTD	Revenue YoY %
FY2018	**$23,860,891.17**	**$23,860,891.17**	
2017 Jul	$1,423,357.32	$1,423,357.32	
2017 Aug	$2,057,902.45	$3,481,259.78	
2017 Sep	$2,523,947.55	$6,005,207.32	
2017 Oct	$561,681.48	$6,566,888.80	
2017 Nov	$4,764,920.16	$11,331,808.96	
2017 Dec	$596,746.56	$11,928,555.52	
2018 Jan	$1,327,674.63	$13,256,230.15	
2018 Feb	$3,936,463.31	$17,192,693.45	
2018 Mar	$700,873.18	$17,893,566.64	
2018 Apr	$1,519,275.24	$19,412,841.88	
2018 May	$2,960,378.09	$22,373,219.97	
2018 Jun	$1,487,671.19	$23,860,891.17	
FY2019	**$34,070,108.50**	**$34,070,108.50**	**42.79%**
2018 Jul	$2,939,691.00	$2,939,691.00	106.53%
2018 Aug	$3,964,801.20	$6,904,492.20	92.66%
2018 Sep	$3,287,605.93	$10,192,098.13	30.26%
2018 Oct	$2,157,287.40	$12,349,385.53	284.08%
2018 Nov	$3,611,092.23	$15,960,477.76	-24.22%
2018 Dec	$2,624,078.39	$18,584,556.15	339.73%
2019 Jan	$1,847,691.91	$20,432,248.06	39.17%
2019 Feb	$2,829,361.64	$23,261,609.70	-28.12%
2019 Mar	$2,092,434.35	$25,354,044.05	198.55%
2019 Apr	$2,405,970.99	$27,760,015.05	58.[illegible]%
2019 May	$3,459,444.04	$31,219,459.08	1[illegible]
2019 Jun	$2,850,649.42	$34,070,108.50	91.62%

> **Note**
>
> The **Revenue YoY %** measure demonstrates a good use of DAX variables. The measure improves the readability of the formula and allows you to unit test part of the measure logic (by returning the **RevenuePriorYear** variable value). Additionally, the measure is an optimal formula because it doesn't need to retrieve the prior year's revenue value twice. Having stored it once in a variable, the `RETURN` clause uses to the variable value twice.

100 XP

Additional time intelligence calculations

5 minutes

Other DAX time intelligence functions exist that are concerned with returning a single date. You'll learn about these functions by applying them in two different scenarios.

The FIRSTDATE and the LASTDATE DAX functions return the first and last date in the current filter context for the specified column of dates.

Calculate new occurrences

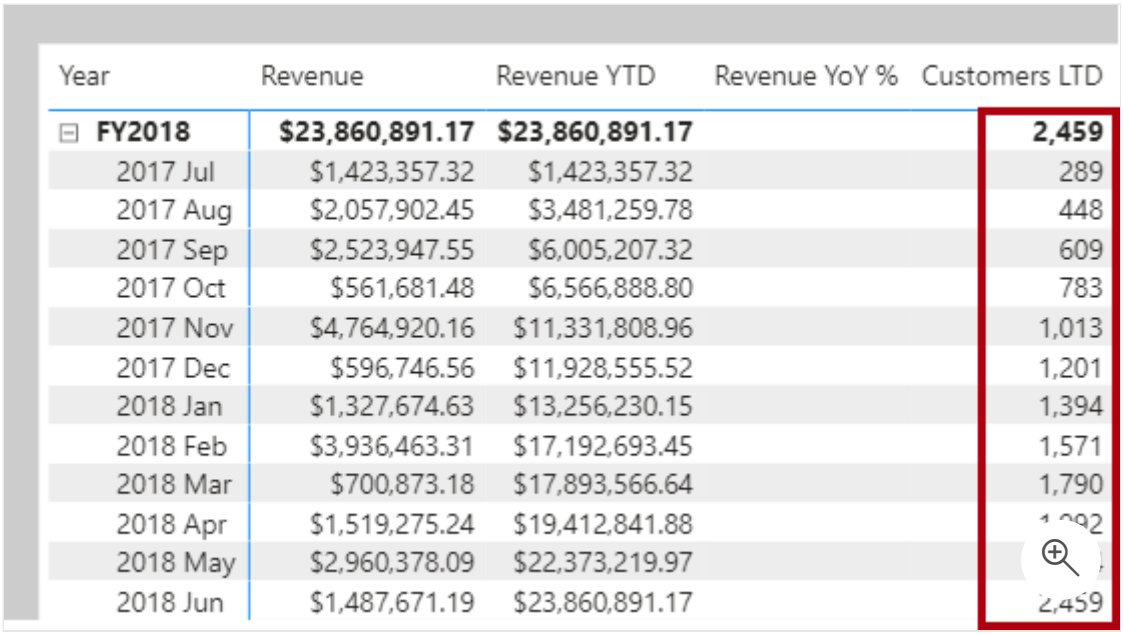
Another use of time intelligence functions is to count new occurrences. The following example shows how you can calculate the number of new customers for a time period. A new customer is counted in the time period in which they made their first purchase.

Your first task is to add the following measure to the **Sales** table that counts the number of distinct customers *life-to-date* (LTD). Life-to-date means from the beginning of time until the last date in filter context. Format the measure as a whole number by using the thousands separator.

DAX

```
Customers LTD =
VAR CustomersLTD =
    CALCULATE(
        DISTINCTCOUNT(Sales[CustomerKey]),
        DATESBETWEEN(
            'Date'[Date],
            BLANK(),
            MAX('Date'[Date])
        ),
        'Sales Order'[Channel] = "Internet"
    )
RETURN
    CustomersLTD
```

Add the **Customers LTD** measure to the matrix visual. Notice that it produces a result of distinct customers LTD until the end of each month.



Year	Revenue	Revenue YTD	Revenue YoY %	Customers LTD
FY2018	**$23,860,891.17**	**$23,860,891.17**		**2,459**
2017 Jul	$1,423,357.32	$1,423,357.32		289
2017 Aug	$2,057,902.45	$3,481,259.78		448
2017 Sep	$2,523,947.55	$6,005,207.32		609
2017 Oct	$561,681.48	$6,566,888.80		783
2017 Nov	$4,764,920.16	$11,331,808.96		1,013
2017 Dec	$596,746.56	$11,928,555.52		1,201
2018 Jan	$1,327,674.63	$13,256,230.15		1,394
2018 Feb	$3,936,463.31	$17,192,693.45		1,571
2018 Mar	$700,873.18	$17,893,566.64		1,790
2018 Apr	$1,519,275.24	$19,412,841.88		1,992
2018 May	$2,960,378.09	$22,373,219.97		
2018 Jun	$1,487,671.19	$23,860,891.17		2,459

The `DATESBETWEEN` function returns a table that contains a column of dates that begins with a given start date and continues until a given end date. When the start date is BLANK, it will use the first date in the date column. (Conversely, when the end date is BLANK, it will use the last date in the date column.) In this case, the end date is determined by the MAX function, which returns the last date in filter context. Therefore, if the month of August 2017 is in filter context, then the MAX function will return August 31, 2017 and the `DATESBETWEEN` function will return all dates through to August 31, 2017.

Next, you will modify the measure by renaming it to **New Customers** and by adding a second variable to store the count of distinct customers *before* the time period in filter context. The `RETURN` clause now subtracts this value from LTD customers to produce a result, which is the number of new customers in the time period.

DAX

```
New Customers =
VAR CustomersLTD =
    CALCULATE(
        DISTINCTCOUNT(Sales[CustomerKey]),
        DATESBETWEEN(
            'Date'[Date],
            BLANK(),
            MAX('Date'[Date])
        ),
    'Sales Order'[Channel] = "Internet"
    )
VAR CustomersPrior =
    CALCULATE(
        DISTINCTCOUNT(Sales[CustomerKey]),
        DATESBETWEEN(
            'Date'[Date],
            BLANK(),
            MIN('Date'[Date]) - 1
```

```
        ),
        'Sales Order'[Channel] = "Internet"
    )
RETURN
    CustomersLTD - CustomersPrior
```

Year	Revenue	Revenue YTD	Revenue YoY %	New Customers
FY2018	**$23,860,891.17**	**$23,860,891.17**		**2,459**
2017 Jul	$1,423,357.32	$1,423,357.32		289
2017 Aug	$2,057,902.45	$3,481,259.78		159
2017 Sep	$2,523,947.55	$6,005,207.32		161
2017 Oct	$561,681.48	$6,566,888.80		174
2017 Nov	$4,764,920.16	$11,331,808.96		230
2017 Dec	$596,746.56	$11,928,555.52		188
2018 Jan	$1,327,674.63	$13,256,230.15		193
2018 Feb	$3,936,463.31	$17,192,693.45		177
2018 Mar	$700,873.18	$17,893,566.64		219
2018 Apr	$1,519,275.24	$19,412,841.88		202
2018 May	$2,960,378.09	$22,373,219.97		
2018 Jun	$1,487,671.19	$23,860,891.17		245

For the **CustomersPrior** variable, notice that the `DATESBETWEEN` function includes dates until the first date in filter context *minus* one. Because Microsoft Power BI internally stores dates as numbers, you can add or subtract numbers to shift a date.

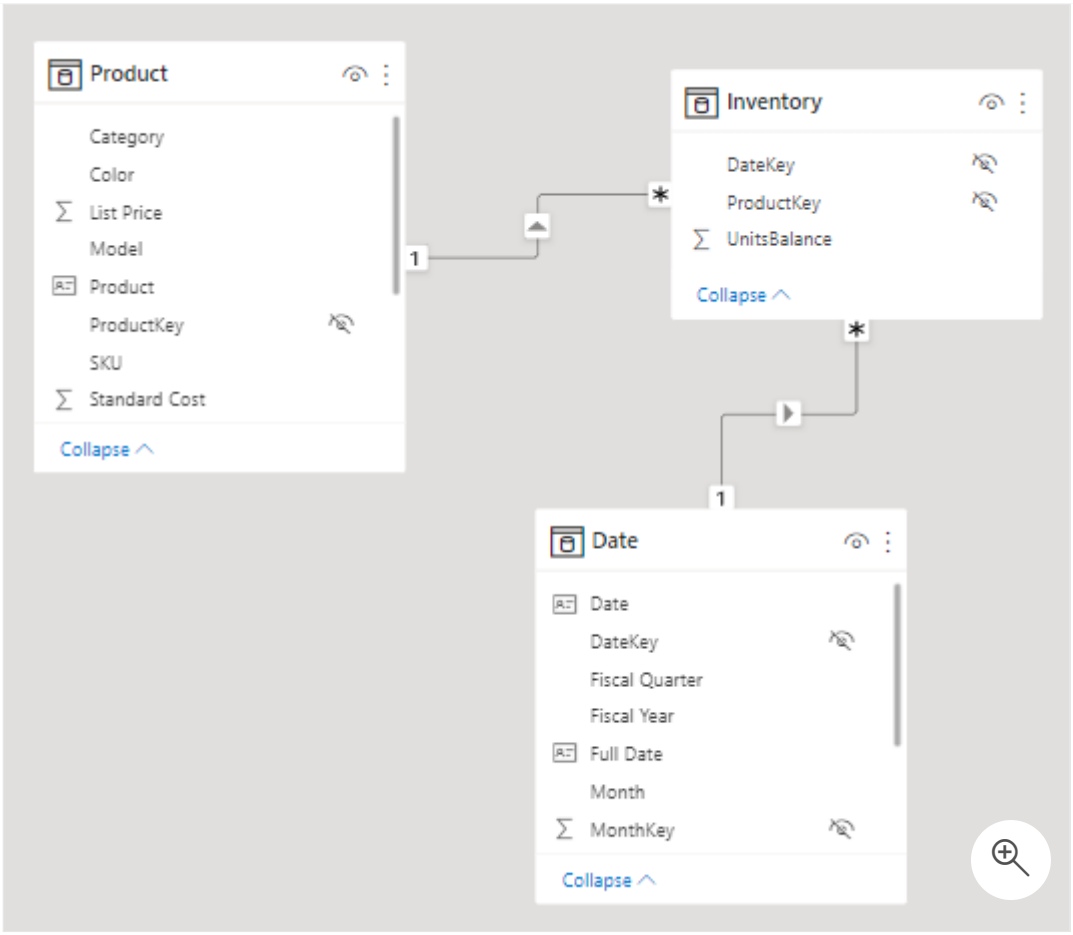
Snapshot calculations

Occasionally, fact data is stored as snapshots in time. Common examples include inventory stock levels or account balances. A snapshot of values is loaded into the table on a periodic basis.

When summarizing snapshot values (like inventory stock levels), you can summarize values across any dimension except date. Adding stock level counts across product categories produces a meaningful summary, but adding stock level counts across dates does not. Adding yesterday's stock level to today's stock level isn't a useful operation to perform (unless you want to average that result).

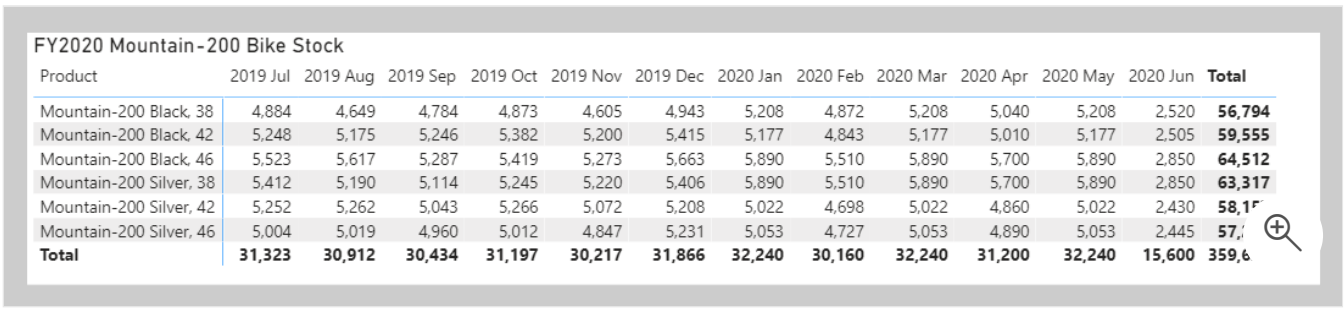
When you are summarizing snapshot tables, measure formulas can rely on DAX time intelligence functions to enforce a single date filter.

In the following example, you will explore a scenario for the Adventure Works company. Switch to model view and select the **Inventory** model diagram.



Notice that the diagram shows three tables: **Product**, **Date**, and **Inventory**. The **Inventory** table stores snapshots of unit balances for each date and product. Importantly, the table contains no missing dates and no duplicate entries for any product on the same date. Also, the last snapshot record is stored for the date of June 15, 2020.

Now, switch to report view and select **Page 2** of the report. Add the **UnitsBalance** column of the **Inventory** table to the matrix visual. Its default summarization is set to sum values.



FY2020 Mountain-200 Bike Stock

Product	2019 Jul	2019 Aug	2019 Sep	2019 Oct	2019 Nov	2019 Dec	2020 Jan	2020 Feb	2020 Mar	2020 Apr	2020 May	2020 Jun	Total
Mountain-200 Black, 38	4,884	4,649	4,784	4,873	4,605	4,943	5,208	4,872	5,208	5,040	5,208	2,520	**56,794**
Mountain-200 Black, 42	5,248	5,175	5,246	5,382	5,200	5,415	5,177	4,843	5,177	5,010	5,177	2,505	**59,555**
Mountain-200 Black, 46	5,523	5,617	5,287	5,419	5,273	5,663	5,890	5,510	5,890	5,700	5,890	2,850	**64,512**
Mountain-200 Silver, 38	5,412	5,190	5,114	5,245	5,220	5,406	5,890	5,510	5,890	5,700	5,890	2,850	**63,317**
Mountain-200 Silver, 42	5,252	5,262	5,043	5,266	5,072	5,208	5,022	4,698	5,022	4,860	5,022	2,430	**58,1**
Mountain-200 Silver, 46	5,004	5,019	4,960	5,012	4,847	5,231	5,053	4,727	5,053	4,890	5,053	2,445	**57,**
Total	**31,323**	**30,912**	**30,434**	**31,197**	**30,217**	**31,866**	**32,240**	**30,160**	**32,240**	**31,200**	**32,240**	**15,600**	**359,6**

This visual configuration is an example of how not to summarize a snapshot value. Adding daily snapshot balances together doesn't produce a meaningful result. Therefore, remove the **UnitsBalance** field from the matrix visual.

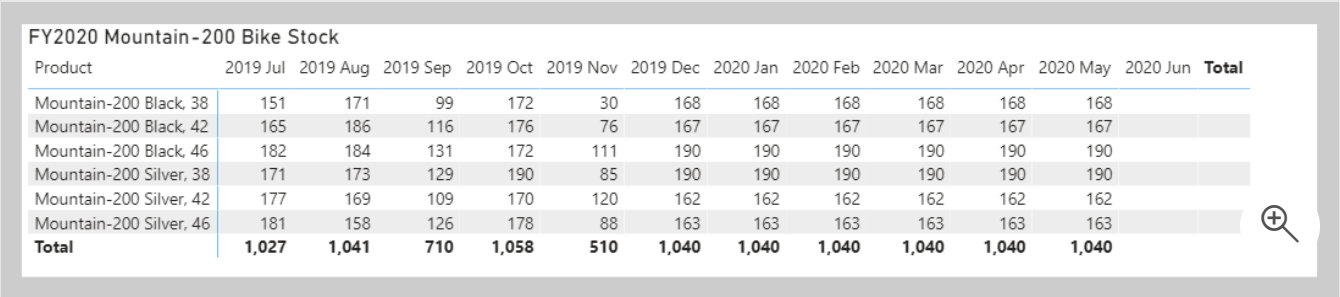
Now, you'll add a measure to the **Inventory** table that sums the **UnitsBalance** value *for a single date*. The date will be the last date of each time period. It's achieved by using the `LASTDATE` function. Format the measure as a whole number with the thousands separator.

DAX

```
Stock on Hand =
CALCULATE(
    SUM(Inventory[UnitsBalance]),
    LASTDATE('Date'[Date])
)
```

> **Note**
>
> Notice that the measure formula uses the `SUM` function. An aggregate function must be used (measures don't allow direct references to columns), but given that only one row exists for each product for each date, the `SUM` function will only operate over a single row.

Add the **Stock on Hand** measure to the matrix visual. The value for each product is now based on the last recorded units balance for each month.



FY2020 Mountain-200 Bike Stock

Product	2019 Jul	2019 Aug	2019 Sep	2019 Oct	2019 Nov	2019 Dec	2020 Jan	2020 Feb	2020 Mar	2020 Apr	2020 May	2020 Jun	Total
Mountain-200 Black, 38	151	171	99	172	30	168	168	168	168	168	168		
Mountain-200 Black, 42	165	186	116	176	76	167	167	167	167	167	167		
Mountain-200 Black, 46	182	184	131	172	111	190	190	190	190	190	190		
Mountain-200 Silver, 38	171	173	129	190	85	190	190	190	190	190	190		
Mountain-200 Silver, 42	177	169	109	170	120	162	162	162	162	162	162		
Mountain-200 Silver, 46	181	158	126	178	88	163	163	163	163	163	163		
Total	**1,027**	**1,041**	**710**	**1,058**	**510**	**1,040**	**1,040**	**1,040**	**1,040**	**1,040**	**1,040**		

The measure returns BLANKs for June 2020 because no record exists for the last date in June. According to the data, it hasn't happened yet.

Filtering by the last date in filter context has inherent problems: A recorded date might not exist because it hasn't yet happened, or perhaps because stock balances aren't recorded on weekends.

Your next step is to adjust the measure formula to determine the last date *that has a non-BLANK result* and then filter by that date. You can achieve this task by using the LASTNONBLANK DAX function.

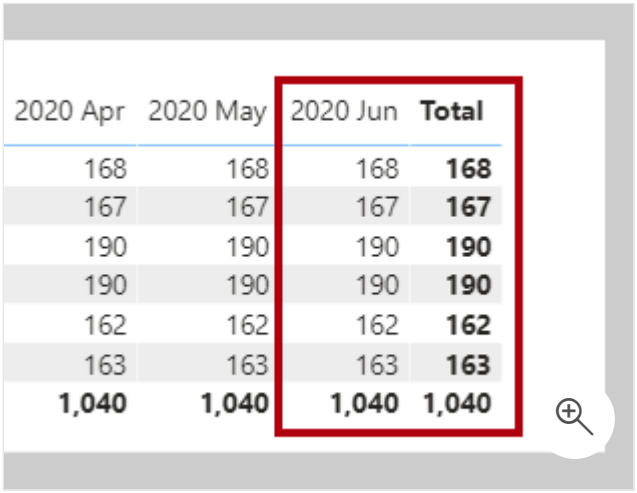
Use the following measure definition to modify the **Stock on Hand** measure.

DAX

```
Stock on Hand =
CALCULATE(
    SUM(Inventory[UnitsBalance]),
    LASTNONBLANK(
        'Date'[Date],
        CALCULATE(SUM(Inventory[UnitsBalance]))
```

```
        )
)
```

In the matrix visual, notice the values for June 2020 and the total (representing the entire year).



2020 Apr	2020 May	2020 Jun	**Total**
168	168	168	**168**
167	167	167	**167**
190	190	190	**190**
190	190	190	**190**
162	162	162	**162**
163	163	163	**163**
1,040	**1,040**	**1,040**	**1,040**

The `LASTNONBLANK` function is an iterator function. It returns the last date that produces a non-BLANK result. It achieves this result by iterating through all dates in filter context *in descending chronological order*. (Conversely, the `FIRSTNONBLANK` iterates in ascending chronological order.) For each date, it evaluates the passed in expression. When it encounters a non-BLANK result, the function returns the date. That date is then used to filter the `CALCULATE` function.

> **Note**
>
> The `LASTNONBLANK` function evaluates its expression in row context. The `CALCULATE` function must be used to transition the row context to filter context to correctly evaluate the expression.

You should now hide the **Inventory** table **UnitsBalance** column. It will prevent report authors from inappropriately summarizing snapshot unit balances.

Next unit: Exercise - Create Advanced DAX Calculations in Power BI Desktop

Continue >

200 XP

Check your knowledge

3 minutes

Answer the following questions to see what you've learned.

1. In the context of data model calculations, which statement best describes time intelligence? *

- ○ Snapshot balance reporting
- ○ Filter context modifications involving a date table

✔ **Time intelligence calculations modify date filter contexts.**

- ○ Complex calculations involving time

✘ **Time intelligence calculations aren't necessarily complex.**

- ○ Calculations involving hours, minutes, or seconds

2. You're developing a data model in Power BI Desktop. You've just added a date table by using the CALENDARAUTO function. You've extended it with calculated columns, and you've related it to other model tables. What else should you do to ensure that DAX time intelligence calculations work correctly? *

- ○ Add time intelligence measures to the date table.
- ○ Mark as a Date table.

✔ **You must mark the date table so that Power BI can correctly filter its dates.**

- ○ Add a fiscal hierarchy.
- ○ Add a date column.

3. You have a table that stores account balance snapshots for each date, excluding weekends. You need to ensure that your measure formula only filters by a single date. Also,

if no record is on the last date of a time period, it should use the latest account balance. Which DAX time intelligence function should you use? *

- ○ FIRST
- ○ FIRSTNONBLANK
- ○ LAST

✗ The LAST function will return the first date in the filter context. This option won't help you achieve the objective. The requirement is to filter by the last date where a snapshot record exists.

- ○ LASTNONBLANK

✓ The LASTNONBLANK function will return the last date in the filter context where a snapshot record exists. This option will help you achieve the objective.

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

Practice Assessment for Exam PL-300: Microsoft Power BI Data Analyst

Question 34 of 50

You create a data model in Power BI Desktop that contains DAX calculated columns and measures. You now need to create a report.

In which two places can a DAX calculated column be used, but a DAX calculated measure cannot be used? Each correct answer presents a complete solution.

- [ ] as a filter in the "Filters on this page" well of the Filters pane

✓This answer is correct.

- [ ] as a filter in the "Filters on this visual" well of the Filters pane

This answer is incorrect.

- [ ] as an item in the "Add drill-through fields here" well of the Visualizations pane

This answer is incorrect.

- [ ] as an item in the Fields well of a slicer

✓This answer is correct.

Unlike a measure, a calculated column can be used in a slicer to place filter options on the report page. DAX measures cannot be placed in the "Filters on this page" well. They can only be placed per visual, in the "Filters on this visual" well of the Filters Pane. Both DAX columns and measures may be used as a visual-level filter. Both DAX columns and measures can be used in the drillthrough well.
Introduction to DAX - Training | Microsoft Learn

Next >

Check Your Answer

Practice Assessment for Exam PL-300: Microsoft Power BI Data Analyst

Question 35 of 50

You need to create a row level security (RLS) role for a dataset.

What should you do first?

- ◯ Assign workspace members to the security group in the Power BI service.
- ◯ In Power BI Desktop, open Manage roles.

✓This answer is correct.

- ◯ In Power BI Desktop, open the Advanced Editor in the Power Query Editor.
- ◯ In the Power BI service, open the security settings for the dataset.

RLS roles are created or modified from the Manage roles space. You can assign AD users/groups to an existing role in the security settings, but new RLS roles must be created in Power BI desktop or other model authoring external tool. RLS configurations cannot be accessed from any Power Query window. Before you can assign users to a security group, it first needs to be created for the model in Power BI Desktop.

Row-level security (RLS) with Power BI - Power BI | Microsoft Learn

Design a data model in Power BI - Training | Microsoft Learn

Next >

Check Your Answer

Practice Assessment for Exam PL-300: Microsoft Power BI Data Analyst

Question 36 of 50

You need to develop a quick measure in Power BI Desktop.

Which two elements can you use? Each correct answer presents a complete solution.

- [x] Calculations

✓This answer is correct.

- [ ] Conditional columns
- [ ] Data Analysis Expression (DAX) queries

This answer is incorrect.

- [x] Fields

✓This answer is correct.

- [ ] Power Query M functions

This answer is incorrect.

When creating a quick measure in Power BI Desktop, you apply calculations to fields. You do not explicitly create a DAX query, but you choose calculations and fields, which result in automatic generation of a DAX query. Conditional columns are separate from quick measures. Unlike quick measures, they create a value for each row in a table and are stored in the .pbix file. Power Query M functions are not directly accessible from the Quick Measure interface.

Lab - Model data in Power BI Desktop, Part 1 - Training | Microsoft Learn

Introduction to DAX - Training | Microsoft Learn

Practice Assessment for Exam PL-300: Microsoft Power BI Data Analyst

Question 37 of 50

You have a Power BI Desktop dataset that includes a table named Warehouse. The Warehouse table includes a column named Inventory Count, which contains the current number of items for each row of a particular type on a given day.

You have the following Data Analysis Expression (DAX) query that calculates the sum of all values in the Inventory Count column of the Warehouse table:

```
Current Inventory Count =
CALCULATE (
    SUM ( 'Warehouse'[Inventory Count] ))
```

You need to ensure that `Current Inventory Count` returns only the current total number of inventory items, rather than the sum of all inventory items that includes item counts from previous days.

What DAX function should you include in the query?

- ○ `CALENDAR`
- ○ `CALENDARAUTO`
- ○ `DISTINCTCOUNT`
- ◉ `LASTDATE`

✓ **This answer is correct.**

The `LASTDATE` function will ensure that the `SUM` function applies only to the last date of the time period, resulting in a semi-additive behavior. The `DISTINCTCOUNT` function counts the number of distinct values in a column, which results in additive behavior. The `CALENDAR` function returns a table with a column named

Date that contains a contiguous set of dates based on the start date and end date that you specify. The `CALENDARAUTO` function returns a table with a column named Date that contains a contiguous set of dates based on data in the model.

Create semi-additive measures - Training | Microsoft Learn

Next >

Check Your Answer

You have a Power BI Desktop model.

You need to determine when to use implicit and explicit measures.

What is a feature of an implicit measure that explicit measure does **NOT** have?

- ◯ End-users can change the aggregation type of implicit measure from the Values well of a visual.

✓**This answer is correct.**

- ◯ Implicit measures can be used as a Drillthrough field.
- ◯ Implicit measures can be used to create Quick measures.
- ◯ Implicit measures can be used with Field Parameters.

Implicit measures can select from one of nine aggregations when placed in the Values well of a visual. Both Implicit and Explicit measures can be used as a Drillthrough field, to create quick measures, and with Field Parameters.
Introduction - Training | Microsoft Learn

Next > **Check Your Answer**

Practice Assessment for Exam PL-300: Microsoft Power BI Data Analyst

Question 39 of 50

You need to reduce the size of a Power BI model that contains two dimension tables named Date and Location, and one fact table named Temperatures. The Temperatures table contains the following fields:

- Reading Time (datetime)
- DateKey (date)
- LocationKey (whole number)
- Temp C (decimal)

You have one row for every 5-minute interval for each location. The Temperatures table is related to the Date and Location dimensions by using many-to-one relationships.

You need to reduce the cardinality of the table. The solution must ensure that the dataset supports reports that analyze average temperature by hour and location.

What two actions should you perform? Each correct answer presents part of the solution.

- [x] Create a column that contains the time values for the start of the hour of the Reading Time value.

✓This answer is correct.

- [x] Disable the query load on the Temperatures query in the Power Query Editor.

This answer is incorrect.

- [ ] Remove the rows that occur exactly at 0 minutes and 0 seconds on the hour.
- [x] Use the Group By functionality to aggregate the rows by hour, DateKey, and LocationKey and then create an average Temp C value per row.

✓This answer is correct.

- [ ] Use the Group By functionality to aggregate the rows by DateKey, Reading Time, and LocationKey and then create a max Temp C value per row.

Creating a column that displays the hour is necessary to summarize by hour when there is no Time or Hour dimension. Summarizing the table by using the Group By functionality reduces the number of rows. Reporting requirements dictate that the aggregated temperature value should be an average. The table should be summarized by hour, DateKey, and LocationKey in order to support the required reports. Removing rows that occur exactly on the hour removes data and does not support the reporting requirements. Disabling the query load removes the table from the model completely.

Reduce cardinality - Training | Microsoft Learn

Next >

Check Your Answer

Practice Assessment for Exam PL-300: Microsoft Power BI Data Analyst

Question 40 of 50

You plan to run Power BI Desktop Performance Analyzer.

You need to ensure that the data engine cache will NOT impact the test results without restarting Power BI Desktop.

What should you do first?

- ◯ Add a blank page to the .pbix file and select it.
- ◯ Connect DAX Studio to the data model.

✓This answer is correct.

- ◯ Invoke the Clear Cache function.
- ◯ Invoke the Refresh Metadata function.

DAX Studio, once connected to the data model, can be used to clear the data engine cache. The Clear Cache function can be invoked from DAX Studio, once you connect it to the data model. The Refresh Metadata function can be invoked from DAX Studio to update the metadata of the currently selected model. Adding a blank page to the .pbix file and selecting it is the first step in clearing the visual cache, not the data engine cache.

Review performance of measures, relationships, and visuals - Training | Microsoft Learn

Next >

Check Your Answer